Appendix 1

DESIGN AND MAINTENANCE SPECIFICATION FOR CTG REACHABILITY & CONTROL SUBSYSTEMS

Kevin Harer

Design and Maintenance Specification

FOR

CTG

Reachability & Control Subsystems

Revision 1.0

Date:02/14/00

AUTHOR(S):, Kevin Harer

1. Introduction

1.1. Document Scope

This document serves as a design and maintenance document for the Reachability & Control Subsystem of the "Coverage-based Test Generation" product (CTG). This document describes various mechanisms, techniques, and policies which are used to coordinate formal and informal engines as a solution to our specific reachability problem. This includes interfaces to the engines and integration techniques for those engines.

This document does not describe in detail individual engines or underlying infrastructure of the formal analysis environment. The document does not describe detailed User Interface issues, mechanisms, methodologies, or flows. VERA environmental modeling and compilation is not discussed. Refer to appropriate documentation for those other technologies.

1.2. Intended Audience

The intended audience is technical engineering staff which are actively involved in CTG development. The document may be useful for non-technical persons who are familiar with CTG problems and issues.

1.3. Document Organization

The remainder of this document is organized in the following way. Section 2 provides an overview of the problem CTG addresses and the general approach from a users perspective. Section 3 provides an architectural overview of the CTG system implementation. Section 4 provides detailed descriptions of mechanisms used to coordinate, control, and/or integrates the various engines and UI into a cohesive system. Section 5 provides detailed descriptions of goal creation and maintenance in CTG. Section 6 provides detailed interfacing mechanisms for the UI process. Section 7 provides detailed descriptions of the interface with the explicit states simulator VCS, as well as detailed explanation of test generation and replay mechanisms and capabilities.

2. CTG Usage Overview

The basic problem that CTG attacks is the following. Given some Design Under Test (DUT), and some goal states of that design, classify all goals as either Unreachable or Reached. If a goal is classified Unreachable, it must be proven with respect to some start state and DUT environmental conditions. If a goal is Reached, then CTG must save a simulation test suitable to reach that goal in some later VCS stand-alone session. Any goal which is not classified as Reached or Unreachable is said to be Unknown; CTG attempts to minimize the number of these Unknown goals.

CTG uses a state coverage metric, hence goals are states, or sets of states, of the design These states are defined by a small number of "coverage variables". These coverage variables are on the order of dozens, while the total latches in the design are in the thousands. Each goal is a different combination of these coverage variables. As an example; given coverage variables {c1, c2}, CTG attempts to reach "states" {00}, {01}, {10}, {11}. Note that these "states" are technically "cubes", but we call them states.

The CTG objective then is to classify as many different combinations of these coverage variables as possible. This task is known as "Coverage Driven Test Generation".

The method CTG uses to "reach" (or cover) goals is to interleave classic explicit state simulation (e.g. VCS) with formal exhaustive searches using some explicit design states as starting points for the search. If a formal search is successful, then the explicit state simulator is used to build up a simulatable test which reaches the goal. This scenario is illustrated in Figure 1. In Figure 1, the enclosing box represents the entire state-space of the design. Explicit state simulation is represented by the zig-zag solid line as explicit

states are visited on each successive clock cycle. Stars represent goals which are to be "reached", if possible.

At some time T1, CTG determines that simulation is not reaching new goals. A formal search is used to exhaustively search for a goal from some explicit state, S1, generated by the simulator. This formal search is represented by the concentric rings in Figure 1. When a formal search reaches a goal G1, the sequence of input assignments (termed a "trace") to reach from S1 to G1 are simulated and the goal is then "reached".

This interleaved process continues until all goals are classified or some user-specified resource limit is exceeded (typically time).

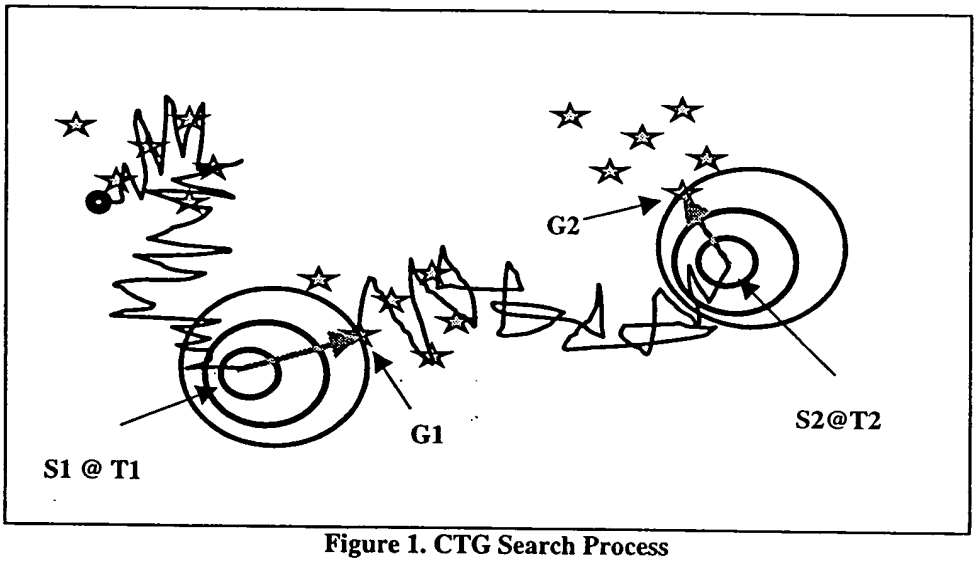


Figure 1. CTG Search Process

Some basic observations which lead to this interleaved engine usage are the following:

- Explicit state simulators are very efficient at reaching "deep" states in the design. Explicit state simulation only reaches a single state at each clock cycle.

- Formal search methods are exhaustive, which means that all reachable simulation states are reached at the same time. Formal reachability algorithms are feasible for short search radius, and commonly intractable for deep search radius.

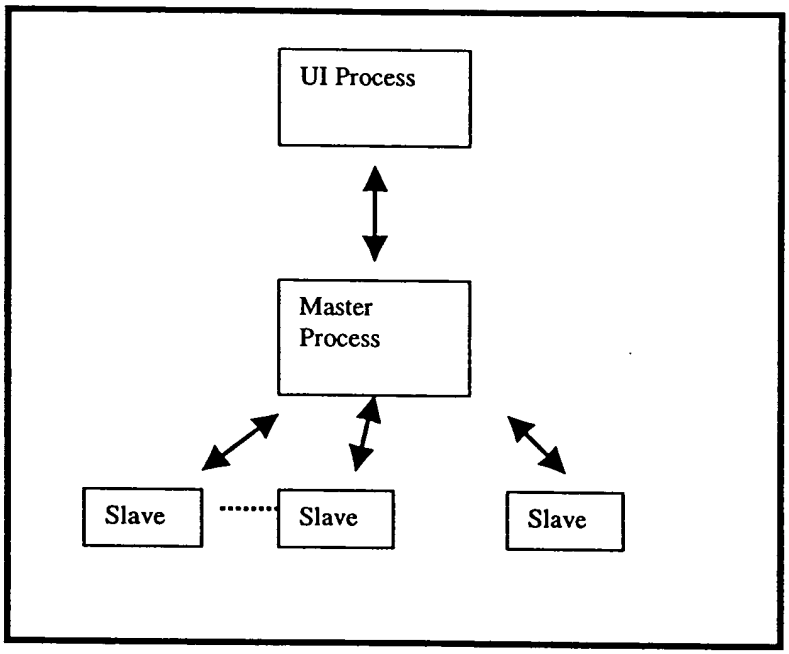
- When a difficult goal is reached, there are often several other goals "close" to that goal in the state-space. This is termed "jackpot" behavior. The rationale for this is due to the nature of the goals that CTG attempts to reach. As described earlier, goals consist of a collection of coverage variables. These coverage variables are the union of smaller control Finite State Machines (FSM's) in the design. Some of these FSM's are relatively easy to transition to new states, others are relatively difficult. When a difficult FSM state is entered, the cross states determined by the easy FSM's are then easily reached; these are the jackpot goals.

3. Architectural Overview

This section provides a general description of large components of the CTG system which accomplish the usage described in the previous section.

3.1. Master/Slave Infrastructure

The overall CTG architecture may best be thought of as a Master/Slave where a Master process interacts and coordinates many concurrent Slave processes. These Slaves are used to "reach" various goals, prove them unreachable, and to interact with the user. Fig.



2 illustrates this architecture.

Figure 2. CTG Process Architecture

As illustrated in Figure 2, a "Master" process is used to coordinate more than 1Slave (or child) processes so that the User Interface process is isolated from all control/coordination problems. The Master process is also given the task of providing communication mechanisms between the Slaves and UI process. In addition, various system heuristics are embodied in the Master process to choose which Slave engine to use at different times in the CTG session.

It is important to note that the Master process is intended to be "live" at all times; the Master process does not "block" for long times while various analysis are carried out by any given Slave. An additional requirement is that the Master process allow Slaves to work concurrently.

The Master process continually polls the Slave and the UI processes for messages and/or commands to be processed. When a message is received from a Slave, it is decoded and either processed by the Master process itself, or passed to the UI. Commands from the UI are processed immediately by the Master process.

The Slave processes of Figure 2 represent various reachability and/or unreachability engines, each running in a separate unix process. Examples of these engines in the CTG system include: symbolic simulation, discrete event simulation (VCS), satisfiability solver, and an image computation engine. Depending on usage, it is possible for 2 or more of these engines to be used in parallel within a single CTG analysis session.

3.2. TCL Control Mechanisms

Control flow for the CTG product is embodied in TCL procedures which execute in the Master process. This TCL control code interacts with Slave processes and the UI process to coordinate all Slaves into a coherent CTG session.

Each Slave process receives commands from the Master process via stdin of the Slave process. The Slave communicates with the Master by writing properly formatted text strings to stdout, or by putting large amounts of data (e.g. BDDs) in data files formatted for the specific task at hand.

Two classes of messages are received by the Master process from Slave processes: formatted and unformatted. Unformatted messages are simply printed to stdout of the Master process. Formatted messages from the Slave are, by definition, intended to be processed by the TCL control program running in Master. The Master process transfers this information from the unix/C world to the TCL world by calling a TCL procedure; this procedure then further decodes the message for handling by the TCL control program. A message "catalog" is maintained such that all pre-defined (thereby formatted) Slave messages are known.

3.3. UI Interfacing

The UI process issues commands to stdin of the Master process, and receives formatted messages from stdout of the Master process. The UI process is responsible for all user presentation of data received from the Master process. This presentation may be either graphic or textual in nature, depending on user needs. A UI message catalog is maintained.

3.4. Reachability Engines

CTG uses a number of reachability engines, each running in its own unix Slave process. Each of these engines use specialized algorithms to search for sequences of inputs which lead from one state (a "start" state) to others ("goal" states). Examples of these reachability engines include:

- Explicit state engine. This is a classic event driven or cycle-based simulator. This engine uses biased, constrained, random simulation techniques to exercise the Design Under Test (DUT) in an attempt to reach goal states. In addition, this engine has the capability to:
  - generate start states for searches performed by other, more "formal", engines,
  - apply sequences of inputs which are generated by the formal engines,
  - save the entire testcase for later re-use by the user
- Symbolic Simulation engine. This engine uses exhaustive (perhaps under-approximated) symbolic simulation methods to attempt to reach goals from specific starting states. If a goal is reached, the engine saves a sequence of

inputs which are replayed by the explicit state engine to prove that the state has been reached.

- Satisfiability Solver engine. The SAT engine uses bounded model checking concepts, together with classic satisfiability techniques, to reach goals from a starting state. If successful, the engine saves a sequence of inputs which are replayed by the explicit state engine to prove that the state has been reached.

3.5. Unreachability Engines

Unreachability engines use various formal techniques to prove that certain goal states are not reachable from some start state. The start state of interest is typically the reset state of the DUT. These Unreachable states are then optionally communicated to the reachability engines to reduce the search problem. Current unreachability engines incude:

- Image Computation Engine. This engine computes an approximate fixed point for reachable states, then inverts the fixed point as a proof of unreachable states. The fixed point computation is typically over-approximated.

3.6. CTG Process Flow

An illustration of the interleaved and concurrent nature of the CTG system is illustrated in Figure 3.

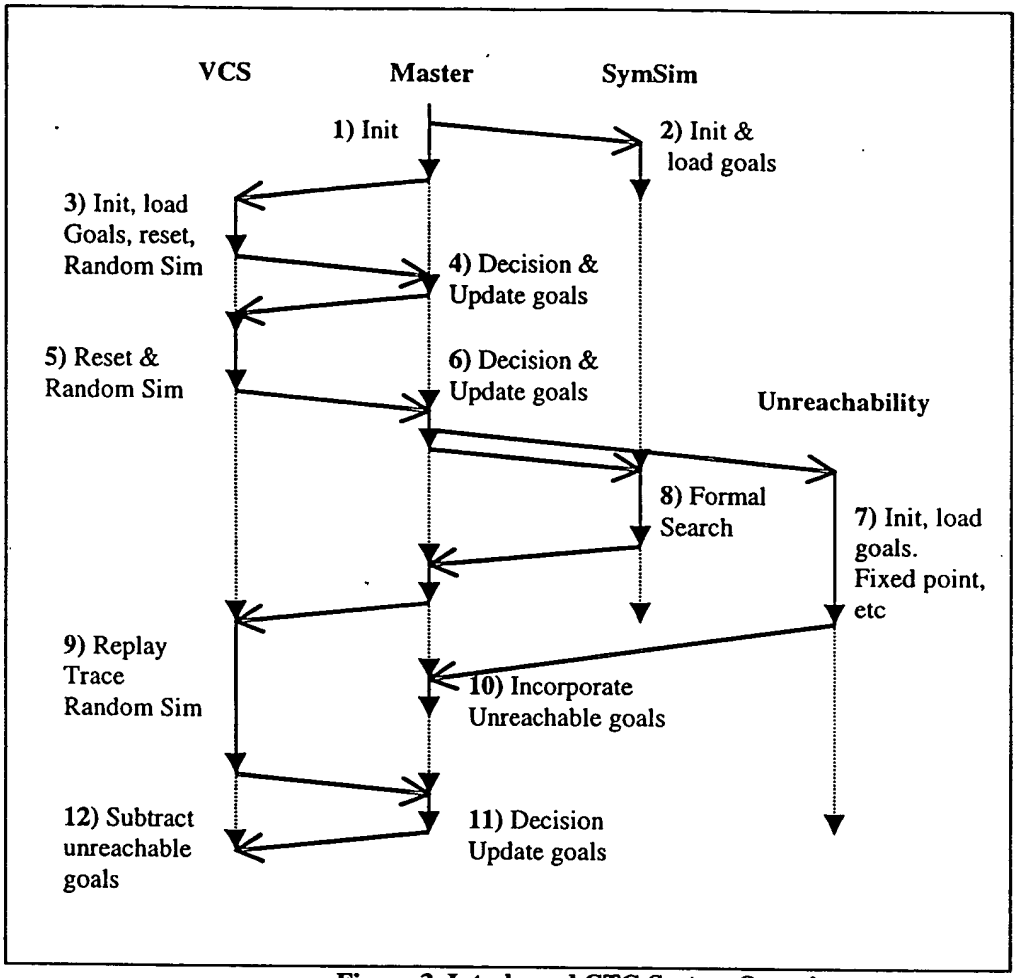


Figure 3. Interleaved CTG System Overview

The diagram of Figure 3 shows an abbreviated, simplified CTG session where VCS, Symbolic Simulation, and Unreachability analysis collaborate to classify goals. The vertical solid arrows indicate that the unix process is "working". The vertical dashed arrows represent that the unix process is waiting for input from some other process before continuing. The horizontal double-arrows indicate messages and/or data being passed between unix processes.

A description of some of the interesting transactions and tasks follows. Note that all UI interactions are omitted, as they are specific to another document.

1. The first thing that happens is that the Master process is invoked. After invocation, the Master process loads HNL, defines goals, and generally does initialization type things.

2. At some point, based on input from the user, the Master process starts and initializes formal reachability engines in separate unix Slave processes. In this example, only the Symbolic Simulation engine is used. The hv program which embodies this reachability engine loads HNL, defines goals, and builds the Symbolic Simulation Manager.

3. The Master process, again under UI direction, starts the VCS Slave process. VCS initializes, loads goals, performs a reset sequence, then runs a number

of random simulation cycles. At this point, the CTG-resident code in VCS is keeping track of which goals are being "reached". At some point, CTG PLI code in VCS determines that "goals are not being reached". VCS stops and returns control to the Master process.

4. The Master process, based on various heuristics and user parameters, makes a decision on how to proceed. In this example, the decision is to perform a reset sequence then do more random simulation in VCS.

5. VCS performs a reset sequence to reset the DUT, then more random simulation occurs. After determining that no more goals are being reached, VCS stops simulating and returns control to the Master process.

6. The Master process decides to perform a formal search, using Symbolic Simulation, from the current design state. This design state is written from VCS and is used to 7) start unreachability analysis, and 8) start a formal search.

7. An Unreachability Slave is started. This is performed by the hv program, which loads HNL and goals, then performs a fixed point computation from the design state provided from VCS. After the fixed point is computed, those goals outside the fixed point are proven unreachable. These unreachable goals are written to disk for later processing by other Slave engines.

8. Simultaneously with unreachability analysis, a formal search is initiated using the Symbolic Simulation engine. This search starts from the design state captured by VCS. In this scenario, the search completes successfully, and the Symbolic Simulator writes out primary input stimulus (i.e. a "trace") which VCS can use to drive the design from the current state to the goal state.

9. VCS replays the trace and hits the new goals. Random simulation continues in an attempt to hit the previously described "jackpot" goals.

10. While VCS is performing random simulation (trace replay, etc), the unreachability engine completes and results are incorporated into the Master process. This unreachable goal data will be communicated to each of the reachability engines on an as-needed basis.

11. At some point, VCS determines that no more goals are being hit, and control is returned to the Master process.

12. The VCS process is sent a message which contains the number of unreachable goals. This count is used to adjust the coverage metric which tells VCS when all goals have been classified.

The CTG session continues interleaving VCS and Symbolic Simulation until all goals are reached. The remainder of this document explains the details about how these things are implemented.

4. Master Process Coordination Mechanisms

As illustrated in Fig. 2, the Master process is the central coordinator of a multi-process concurrent system which attempts to classify goals in some sort of efficient manner. TCL procedures are implemented to

provide key coordination of the Slave engines, and to embody heuristics to attempt to provide an efficient coherent system to the UI. Important infrastructure services are provided in C code packages to provide flexibility to the TCL control programs. The hv program provides this Master process by invoking it the following way:

- hv –I –master
- Typically this invocation is performed by the UI process which 'fork's a unix process, then 'exec's the hv executable in it.

This section presents detailed descriptions of the mechanisms which accomplish the Master process capabilities.

4.1. Process Hierarchy and Communication

The process hierarchy of Figure 2 is the following. The Master process is started by the UI process. Slave processes are started by the Master process. These processes are started using standard unix 'fork' and 'exec' mechanisms.

When a child process is started, the parent process redirects standard-in and standard-out of the child process into unix pipes which are then read and written by the parent process. Thus communication from the parent to the child is accomplished as the child process reads from its stdin file descriptor. The Master and most Slave processes execute the hv program which processes stdin using the Synopsys cci TCL interpreter. The VCS Slave uses the VCS interactive interpreter to read from stdin. Thus all messages to child process must be as either TCL hv commands or VCS commands, depending on the child.

Messages from a child to a parent are communicated via writing to stdout of the child process. The parent process must periodically read from the earlier-mentioned pipe to retrieve and process these messages from the child.

In some cases, where large amounts of data are needed to be communicated between processes, it has proven useful for the data provider to write the data to a temporary file. The consumer is then sent a message saying what/where the data is. The consumer of the data opens the temporary file and processes the data.

4.2. Infrastructure Package (hvCtl)

The hvCtl package provides the key mechanisms to start Slave process, coordinate them, and relay messages to either the UI process or TCL procedures resident in the Master process. Services are provided to allow the TCL programs to communicate and track progress of the Slave processes. These services are accessed exclusively by TCL code which runs in the Master process. This section presents key concepts and commands which access the implementation of those concepts.

4.2.1. Channels

A "Channel" defines a communication path between the Master process and a Slave process. Each Channel has a unique ID which is assigned at the time the Slave process is started. This unique ID is then used to send commands to that Slave and to identify messages received from the Slave. Commands to manipulate and interact with a Channel are the following:

- **hvctl_channel_open** – This command opens a channel by starting a user-specified program in a separate unix process. The Master process then sends messages to the Slave via the hvctl_send_message command.

Messages/responses from the Slave will then be read from stdout of the child process. Upon success, this command assigns a unique ID for for TCL usage to identify the new Channel. Input arguments to the command allow specification of the program to start in the new process, the working directory the process is to run in, and up to 6 arguments to be used in invocation of the program which is started.

- **hvctl_channel_close** – This command closes a channel and cleans up data structures associated with it in the Master process. Input arguments are the Channel ID, and an optional command which terminates the program running in the Slave process.

- **hvctl_channel_send_msg** – This command sends a message to the program running in the Slave process. Input arguments are the Channel ID and a string which is sent to stdin of the child process. The message string must be a syntactically and semantically correct command for the program running in the Slave process. The message is presented to that program on stdin of the Slave process.

- **hvctl_channel_stack_set** – Due to the concurrent, non-blocking nature of the CTG system, a mechanism is needed to allow long-duration Slave commands to execute. The problem for the controlling TCL program then becomes "remembering" what is happening in a Slave process over a long period of time, and what needs to be done when that task is finished. The mechanism provided is termed a "Command Stack", which is described later in this document. This command associates a Command Stack with a specific Channel. Input arguments are the Channel ID and the name of the Command Stack to associate with it.

- **hvctl_channel_stack_get** – This command returns the name of a Command Stack associated with a specific Channel. If no Command Stack has been assigned, the string "<no_stack>" is returned.

- **hvctl_channel_dump_info** – This command is provided for debugging purposes and prints various information about an open Channel.

4.2.2. Engines

An Engine is a "bookkeeping" aid which exists in the Master process, and defines a mechanism which is resident in a Slave. For example, a "Symbolic Simulation Engine" object might exist in the Master process, and correspond to an entire Slave process which is used to perform Symbolic Simulation.

This corresponding Slave mechanism performs some sort of analysis or task which may or may not involve a number of individual commands/responses between Master and Slave. It is possible (and common) to have multiple Engines associated with a single Slave process at the same time. Examples of Engines are the Symbolic Simulator or SAT solver.

Note that the Engine concept is very general, and it has proven useful to have more abstract Engines such as the Goal Manager, the Design-Under-Test Manager, etc.

Engines may have state; for example, the Symbolic Simulation Engine might be in the "RUNNING" state, or the "IDLE" state. This state concept has proven useful to help the controlling TCL program deal with the non-blocking, concurrent nature of CTG.

- **hvctl_engine_type_define** – This command defines a new *type* of Engine. Typically all Engine types are defined at Master process startup.

- **hvctl_engine_state_define** – This command defines a legal state for the newly-defined Engine type. Valid Engine states are typically defined at Master process startup, when the new Engine type is also defined.

- **hvctl_engine_start** – This command creates an *instance* of an Engine type, associated with a specific Channel. The user specifies Engine type, Channel ID, and a name the Engine instance will be known by. The combination of type, Channel, name must be unique.

- **hvctl_engine_state_set**- This command assigns a state to an instance of an Engine. The state must be valid (defined with hvctl_engine_state_define) for the Engine type. If the specified state is invalid, an error occurs. The caller must specify a unique combination of Engine type, name, and Channel ID.

- **hvctl_engine_state_get** - This command returns the current state of a specific Engine. The caller must specify a unique combination of Engine type, name, and Channel ID.

- **hvctl_engine_exists** – This command returns a space-delimited list of names of current Engines which match caller specified parameters. For example, the caller can request "All Engines defined for Channel ID=4", or "All Engines of type Symbolic Simulator". If no Engines are found, the empty string "" is returned.

- **hvctl_engine_stack_set** – This command is analogous to the previous hvctl_channel_stack_set command. The caller specifies a Command Statck, and unique combination of Engine type, name, and Channel ID.

- **hvctl_engine_stack_get** – This command returns the name of the Command Stack associated with a specific Engine. The caller must specify a unique combination of Engine type, name, and Channel ID.

- **hvctl_engine_type_dump** – This is a debugging command which prints various information about a valid Engine type.

- **hvctl_engine_dump_info** – This is a debugging command which prints information about a current Engine instance.

4.2.3. Command Stacks

As mentioned earlier, the non-blocking nature of the Master process leads to an interesting problem when the Master asks the Slave process to execute a time-consuming task. The TCL control program cannot launch a long task, then simply block-and-wait for the task to finish before continuing to the next task in a sequence of tasks which need to be completed. The TCL program must not block; this non-blocking requirement allows for interactions with the UI or other Slaves. Yet somehow, the TCL program needs a convenient mechanism to help remember what to do when the time-consuming Slave task finishes. The Command Stack is the key mechanism provided to assist in this problem.

A Command Stack is a stack of commands which may be attached to either a Channel or an Engine. When a task is finished (detected via a message from the Slave), the next command is popped from the stack and is executed. Common usage by the TCL control program include the following:

- A new Command Stack is created and associated with each Channel when the Channel is opened.

- When a time-consuming task is started by a Slave, a TCL procedure, with necessary arguments, is pushed onto the Command Stack for that Channel. This pushed TCL procedure is to be executed when the time-consuming task is completed.

- Optionally, the state is set on the Engine which will signal completion of the time-consuming task.

- When the task-is-done message is received by the Master process, the command is popped from the Command Stack, and it is executed. Note that this Slave-to-Master message passing is explained later in this document.

The remainder of this section describes commands which directly manipulate Command Stacks.

- **hvctl_command_stack_create** – This command creates a new Command Stack. The caller may optionally assign a name to the Command Stack – if so, the name must be unique.

- **hvctl_command_stack_free** – This command destroys a previously-created Command Stack.

- **hvctl_command_stack_push** – This command pushes a syntactically legal TCL command string onto the command stack. This string is executed at some later point using the TCL "eval" mechanism.

- **hvctl_command_stack_pop** – This command pops the top command string off the Command Stack. If the stack is currently empty, the string "<empty>" is returned.

- **hvctl_command_stack_dump** – This debugging command prints information about a current Command Stack.

4.2.4. Message Poll Loop

As illustrated Figure 2, the Master process needs to continually check for messages from the UI process as well as each of the Slave process. This mechanism uses the unix "select" mechanism to accomplish this. "select" is a unix function which is given a set of open file descriptors to attempt to simultaneously read from. When data is read from one of the files, select returns with the file descriptor of the file with waiting data. The select function has the ability to also "time out" if no file may be read after a certain amount of time.

For the Master process, these files are either stdin of the Master process, or the open "pipe" (another unix-ism) for individual Slave processes. These pipes are created when a Slave process is started, and correspond to stdin and stdout of the Slave process.

When data is read from one of these files, the Master process first determines which file the data is read from. If the file is the output of a Slave process, the text is decoded to determine if the message is formatted in such a way as to be intended for the TCL mechanisms in the Master process. Formatted Slave messages are identified by prepending "HVINFO" and a message ID to the actual data. The message ID is associated with a specific engine and is expected to be understood by the TCL control program in the Master process.

Once the active file and message decoding has been performed, the Master process takes the following action:

- If the data is read from stdin, it is a TCL command from the UI process which needs to be executed. The Synopsys cci command interpreter is used to execute these commands.

- If the data is read from a Slave process, the message is either unformatted or formatted. Unformatted text is relayed to the UI by simply writing the text to stdout of the Master process. Formatted messages are communicated to the TCL control program in Master by calling a predefined TCL procedure. The TCL procedure is called "hv_child_msg_recv" and arguments include Channel ID for the Channel which sent the message, the message ID, and the actual text of the message.

4.3. TCL Coordination Package

The CTG control policies, mechanisms, and heuristics are all implemented in TCL procedures which are executed by the "hv" program running in the Master process. This TCL control code must issue commands to the Slave programs, interact with the UI process, track progress of the test-generation session, etc. This task is complicated, and key aspects are described in this section; it is beyond the scope of this document to explain **all** necessary details.

Note also that this TCL code is dynamic in nature (i.e. it changes often) as heuristics and algorithms change.

4.3.1. Messages and Message Handlers

As previously discussed, when the Master process reads a message from a Slave process, this message may need to be communicated to the TCL control program. This is done by a call to the TCL procedure "hv_child_msg_recv" which includes a unique message ID, the Channel ID the message is from, and the actual message itself.

Messages are defined on an Engine-specific basis, and are used to communicate some data and/or status from a Slave to the TCL control program in the Master process. These messages must obviously be unique, and coordination of message ID/content between the Slave implementation and Master TCL code must be coordinated by the developer of the engine (Slaves issue messages, Master receives & processes them). To help with this coordination, each Engine has a separate TCL procedure where all messages for that Engine are defined and handled. The hv_child_msg_recv procedure decodes message ID enough to decide which Engine-specific message handler to call, then that procedure is called with the message for further processing.

Message Ids are textual, and the collisions are avoided between engines by assigning a unique prefix to all messages for that engine. For example, all messages from the VCS engine are of the form "VCS_<MsgName>" where <MsgName> varies for each message

from the VCS engine. These textual prefixes are documented for all known engines in the hvRecipe.tcl file.

(Developers note: At time of writing this document, message Ids are numeric, and the actual message format includes a data type field. This will soon be modified to reflect the format described in this document.)

To add a new Engine, the developer must perform the following tasks:

1. Create a file <Engine>Handler.tcl and put it in /vobs/propver/src/tcl. This file will implement the Engine-specific message handler TCL procedure hv_<Engine>_child_msg_recv. This file also contains the Engine type and valid states definitions. Allocate a textual message ID prefix for the new Engine, and document it in hvRecipe.tcl.

2. Modify the hv_child_msg_recv procedure in hvRecipe.tcl to recognize your new messages and call your new procedure when a message in the proper range is detected. Source the <Engine>Handler.tcl file where all the other message handlers are sourced, at the end of hvRecipe.tcl.

3. Document your new messages and handle them by implementing the hv_<Engine>_child_msg_recv procedure in the < Engine>Handler.tcl file. Stylistically, use one of the pre-existing *Handler.tcl files in the tcl directory as a template. By convention, this file does not "know" about other engines or high-level procedures. This message handler typically interprets messages, sends notice to the UI, sets engine state, then pops high-level commands off the command stack to continue the CTG session.

4. Instrument the corresponding Slave implementation of the Engine to generate messages in a semantically appropriate manner.

5. Modify the control program TCL code in hvRecipe.tcl to use the new Engine in an appropriate way. This code is discussed in the following section.

At the time of writing of this document, a certain number of Engines have been defined and integrated. Those engines and their usage are:

- Symbolic Simulation Engine. This Engine is implemented in the hv executable and performs Symbolic Simulation from some start state in an attempt to reach some pre-defined goals. If a goal is reached, a "trace" is written for VCS use which applies a sequence of input to drive the design from the start state to the goal state.

- VCS Simulation Engine. This Engine is implemented via PLI code attached to the VCS simulator. This Engine is responsible for observing which goals have been reached as simulation progresses, detecting points at which random simulation should stop, testcase generation and replay, and interfacing with the user-written environmental biasing and constraint code. In addition, this Engine is capable of conveying search start states to the formal reachability engines.

- SAT Engine. This Engine is similar to the Symbolic Simulation Engine except that satisfiability techniques are used to attempt to find a sequence of

inputs from some start state to a goal state. This Engine is implemented in the hv executable.

- Image Computation Engine. This Engine uses symbolic state traversal techniques to prove that certain states are unreachable from specific start states. These provably unreachable states are then used to classify goals which are unreachable. This Engine is implemented in the hv executable.

- Goal Engine. It has proven useful to define a Goal Engine to manage goal creation, update, and coordination between the various Slave process and the Master process . This Engine is implemented in the hv executable.

- DUT Engine. A DUT Engine is used to manage the design load and pre-processing steps which lead up to reachability and/or unreachability analysis. This Engine is implemented in the hv executable.

4.3.2. Session Startup

Refer to a UI-related document for detailed description of user interactions with the CTG system (i.e. "how the User interacts with CTG"). In general, the user will first setup an analysis session by doing things like creating hnl, compiling the simulation model, defining goals, environmental modeling, etc. Following this, the user clicks on "Generate Testcase" and the following sequence of events take place:

- The UI starts the Master process by forking a child process and invoking "hv –I –master" in that process. The UI opens a pipe to read/write stdout/stdin of the Master process.

- The UI issues hv commands to setup the Master process. These commands must accomplish the following:

  - The hvRecipe.tcl file needs to be sourced

  - User-specified session parameters (i.e. TCL knobs) must be set. These parameters are discussed in a later section, but this step basically sets a number of pre-defined TCL variables.

  - The HNL must be loaded into the Master process by calling the hvino_create_hnl_design command.

  - The HvDsg must be created, and any necessary preprocessing must be accomplished. This is done using the hvdsg_create_hv_design command. Note that current HvDsg usage is that both a hierarchical and flat HvDsg are created in the Master process. All formal and analysis is performed using the flat design; the hierarchical design is used for interaction with the user. Commands are provided for mapping between flat and hierarchical worlds.

  - The proper HvMch should be created using hvmch_create_machine.

  - An appropriate cover (HvCic) should be created using hvcic_create_cover

- An SthMgr and OrdMgr must be created using hvsth_create_manager and hvord_create_manager

- The user-specified goals are created in the Master process.

- The UI process requests a Slave process to be started for SAT and/or Symbolic Simulation analysis. This is accomplished by a call to the "hv_setup_formal" TCL procedure. This procedure is defined in hvRecipe.tcl and described later in this document. A Slave process is started, hnl loaded, goals defined, and any algorithm-specific setup is performed (e.g. create the Symbolic Simulation Manager).

- The UI initiates actual test generation by launching VCS in its own Slave process. This is done by calling the TCL procedure: "hv_start_vcs_session". This procedure is defined in hvRecipe.tcl and described later in this document. A Slave process is started and the compiled simulation model (generated by VCS compilation earlier) is started. Goals are loaded in VCS and a reset is performed, then simulation pauses. The Master process then sets up simulation stop conditions (via calls to the CTG PLI code) and initiates a simulation run from the reset state.

- At this point, the CTG PLI code within the simulator is recording goals which are reached, as well as any information needed to later write out the final testcase.

From this point on, the CTG session progresses according to the "Phase Progression" described in the following section.

4.3.3. CTG Analysis "Phase" Progression

After the reachability session is initiated, the resultant actions are highly dependent on various heuristics, user-specified direction, and how "hard" the design and/or goalset is. In general, the session progresses through a number of phases or modes. These phases are designed to in general be progressive in CPU and Memory cost.

This section describes the phase progression. Search Engine usage in those phases are described as are exit conditions for the phase. Note that this progression is the heart of CTG and can be expected to change **"A LOT"** as CTG is exposed to more designs and the development team learns how to effectively use all the technology. This section is provided to give the reader a feel for how things worked at least once in the past.

A CTG reachability session transitions through the following phases in order. At any point, if all goals have been reached, then the testcase is written and the system stops. In the following discussion, certain user settings (i.e. "Knobs") are referenced; these knobs are cataloged and described elsewhere in this document. In the following discussion, knobs are italicized and in bold.

- MODE_INIT

This is the initial mode that CTG starts in. The VCS Engine is used to repeatedly reset the design followed by a number of biased random simulation cycles. The number of times to repeat this sequence is specified by ***initRunsMax*** and the number of simulation cycles to run is specified by ***initCycles***. ***initCycles*** specifies that VCS will stop when this many system clock cycles are seen without hitting a previously unreached goal.

When this initial run-from-reset phase is finished, VCS performs a final reset then stops. The reset state is written to a file by VCS. If requested by the user by setting the ***unreach*** knob to "UNREACH_LMBM", an unreachability analysis is initiated.

If ***satCycMax*** is > 0, CTG progresses to MODE_INIT_SAT, else MODE_INIT_SYM.

- MODE_INIT_SAT

In this phase, CTG uses the SAT engine to search for goals from the reset state. SAT uses the resource limits ***satCycMax*** and ***satCpuMax*** to know how hard to search.

If a goal is reached, the trace is replayed in VCS followed by random simulation until **initCycles** system clock cycles are seen without hitting a new goal. Then VCS resets and waits for another formal search to be carried out.

If SAT fails to reach a goal from the reset state, then CTG progresses to the next phase. If ***symsimCycMax*** > 0, then MODE_INIT_SYM is entered. Otherwise CTG enters MODE_SAT.

- MODE_INIT_SYM

This phase is entirely the same as MODE_INIT_SAT, except the symbolic simulation search engine is used. Resource limits are defined by ***symsimCpuMax*** and ***symsimCycMax*** knobs. Upon a formal search miss, CTG switches to the next phase. If ***satCycMax*** > 0, then MODE_SAT is entered; otherwise MODE_SYMSIM is entered.

- MODE_SAT

In this mode VCS simulates to a deep state (as illustrated in Figure 1) and stops. This deep state is written to file and is used for a formal search using the SAT engine. SAT uses resource limits ***satCycMax*** and ***satCpuMax*** If SAT reaches a goal, the trace is replayed in VCS followed by more random simulation.

In each case, VCS stops to try a formal search after **rsimCycles** are seen without hitting a unreached goal, or when a "fresh" state is seen. The definition and detection of fresh states are described later in this document.

After ***formalMissThresh*** consecutive search misses, VCS is reset. If ***symsimCycMax*** > 0, then MODE_SYMSIM_SAT is entered, else CTG stays in this phase forever.

- MODE_SYMSIM_SAT

This phase is similar to MODE_SAT, in that formal searches are performed from deep states generated by VCS. VCS stops in the same manner as that described in MODE_SAT. The difference is that SAT and Symbolic Simulation searches are alternated. When **formalMissThresh** consecutive formal misses are seen, the design is reset and CTG enters MODE_SYMSIM.

Resource limits for the formal search engines are ***satCycMax***, ***satCpuMax***, ***symsimCpuMax***, and ***symsimCycMax***.

- MODE_SYMSIM

This phase is entirely like MODE_SAT, except that Symbolic Simulation is used instead of SAT.

- MODE_DONE

CTG enters this phase when all goals have been classsified as either unreachable or have been reached.

4.3.4. Externally Used TCL Procedures

This section describes TCL procedures contained in hvRecipe.tcl which are intended to be called by the UI process directly. In addition to calling these TCL procedures, the UI calls many native hv commands (e.g. hvino_create_hnl_design)

4.3.4.1. hv_formal_setup Procedure

The hv_setup_formal TCL procedure performs various setup function needed to start a Slave process with intent to subsequently perform formal reachability. After initiating the Slave process, this procedure returns without waiting for the Slave process to complete setting up. Slave progress is tracked via engine messages handled in the various message handler routines.

Currently supported formal engines are either SAT or Symbolic Simulation. The procedure prototype is the following:

```
hv_setup_formal -design <desName> -type <rchType> \
      -dir <wdPath> -log <logName>
```

This procedure starts the hv program in a Slave process, running in the directory specified by <wdPath>. If –log is specified, all commands issued to the Slave process will be logged in <logName>. The caller must specify the design name to be analyzed, and the type of reachability engine to be used: either "SAT" or "SYMSIM".

After the new channel is opened and a command stack created for it, the Dut and Goal Engines are started for that channel. The Sat or SymSim Engine is also started.

The hv_dut_setup procedure is called to setup the Dut Engine. The hv_load_goals procedure is pushed on the command stack to be called when a message is received from the Slave saying that the HvDsg has been created.

If the Symbolic-Simulator is being setup, the hv_start_symsim_mgr command is pushed on the stack to be executed after goals are loaded.

TCL knobs used by this procedure are the following: hvPath, goalFile, covVarsFile, workDir (if –dir not specified), and verbMode.

4.3.4.2. hv_start_vcs_session Procedure

This procedure is called after all formal reachability engines have been started to start the VCS explicit state simulator. This procedure will start performing random simulation in the MODE_INIT phase described earlier. After random simulation starts, this procedure returns. Procedure prototype is the following

```
hv_start_vcs_session -exe <vcsExe> \
      -map <mapFile> -root <rootModule> \
      -dir <wdPath> -log <logName>
```

This procedure starts a new channel, attaches a command stack to it, then executes the compiled VCS simulator in it (given by the path <vcsExe>). Depending on knob

settings and use of the –log switch, VCS may be started using a number of switches to that program itself.

<rootModule> is required and is the verilog module which is being analyzed in the formal engines. <mapFile> is required and specifies verilog reg's which are mapped to SEQ elements in the HNL, nets in verilog which are derived clocks in HNL (and therefore have corresponding edge detectors), and the net that is the System Clock in HNL. All names in <mapFile> are valid HNL SEQ icell or inet names.

After VCS starts, it performs one reset, it stops and waits for commands (described in a later section of this document). This procedure sends a $startHV PLI command to be processed. The hv_vcs_init_run procedure is pushed on the command stack to be called when $startHV finishes. After these events are initiated, hv_start_vcs_session returns without waiting for anything to complete.

hv_vcs_init_run is popped from the command stack when VCS has invoked, run a reset sequence and stops. This is detected by a VCS_STOP message from VCS, when the VCS engine is in the "initing" state.

TCL knobs used by this procedure are the following: goalFile, workDir (if –dir is not specified), vcsTestFile, outputVcsRchStates.

4.3.4.3. hv_terminate_vcs_session Procedure

This procedure may be called to terminate a test creation session prior to classification of all goals. By default, the test creation initiated by calling the hv_start_vcs_session procedure will continue until **all** goals have been classified. In some cases this takes a long time and the user may wish to terminate the process cleanly and save a testcase which reaches only a portion of all goals. This procedure is used for that eventuality. Procedure prototype is:

```
hv_terminate_vcs_session -force <forceLevel>
```

This procedure sends a command to VCS to stop simulating, write out the testcase requested (when hv_start_vcs_session was called), and wait before exiting the VCS process. By default, VCS will wait until an in-process formal reach attempt has finished. If <forceLevel> ==1 is specified, VCS will terminate immediately without waiting.

4.3.4.4. hv_print_ctg_knobs Procedure

This procedure is implemented in infoFile.tcl where all TCL knobs are defined. This function prints out all current knobs settings. Procedure prototype is:

```
hv_print_ctg_knobs -doc <docRqst>
```

If <docRqst> is non-0, then a 1-line description of knob usage will be printed for each knob.

4.3.5. Key Internal TCL Procedures

This section describes in some detail TCL procedures which are used to carry out a CTG session initiated by the earlier externally used procedures. In general, these procedures are not called by the UI.

4.3.5.1. hv_dut_setup Procedure

This procedure sets up the DUT Engine in a newly-created Slave process running the hv program in Slave mode. Procedure prototype is the fiollowing:

```
hv_dut_setup -chnl <chnlId> -design <dsgName>
```

The state of the dut Engine associated with <chnlId> is set to load_hnl. The "design" TCLvariable is set to <dsgName> in the child process and the "hvino_create_hnl_design" command is issued in the child process. This procedure returns without waiting for the HNL to be loaded.

Completion of hv_dut_setup is detected via a DUT_DSG_DONE message when the flattened HvDsg is done being created. When completion occurs, the next command is popped off the command stack.

TCL knobs used by this procedure are the following: verbMode.

4.3.5.2. hv_load_goals Procedure

This procedure is called to load goals into a newly-created Slave process running the hv program in Slave mode. This procedure runs after hv_dut_setup has processed the design. Procedure prototype is:

```
hv_load_goals -chnl <chnlId> -gfile <goalFile> \
      -var <varsFile> -style <mchStyle>
```

This procedure creates a HvMch, HvSth manager, and HvOrd manager in the Slave associated with <chnlId>. Goals are created by sourcing <goalFile> in that Slave.

As part of HvMch creation, <varsFile> is sourced and expected to create an InetSet named "CovVars" which contains all coverage variables used in any goals. <mchStyle> is one of "COVVAR", "OUTPUTS", "CUTSET", or "MEM" and defines how that HvMch should be created in the Slave process.

- "OUTPUTS" – Create the default HvMch, which prunes the machine against all Primary Outputs of the underlying HNL. This machine is typically used for SAT.

- "COVVAR" – Create the HvMch using the InetSet named "CovVars" to prune the machine against. This machine is typically used for Symbolic Simulation.

- "CUTSET" – Create an HvMch using the InetSet names "CovVars" as the machine outputs, and an InetSet determined by hvcic_compute_inet_cut_set as the inputs to the machine. A cover (HvCic) is created using hvcic_create_cover_from_inet_set using the same cutset InetSet. This machine and cover are typically used for the Image Computation engine.

- "MEM" – Create necessary HvMch's for optimal Symbolic Simulation of a design with embedded memories. A file specified by the memInfoFile knob is sourced in the child process and defines the design-specific memory arrays which will be optimized.

CTG detects that goals are done being loaded by receiving a HVGOL_ENGINE_WORKING message from the goal engine. Data is "DONE" and the goal engine must be in the "load_goalfile" state.

This procedure uses the TCL knobs: verbMode, goalFile, cicMaxLimit, cicBlkSize, and memInfoFile.

4.3.5.3. hv_do_lmbm_unreach Procedure

Thos procedure initiates unreachability analysis using LMBM-style fixed point computation. Note that the underlying design may be a partition over the whole machine (true LMBM) or simply a single block which is a subset of the whole machine (not true LMBM). Procedure prototype is:

```
hv_do_lmbm_unreach -init <initFile> -state <vecName>\
      -design <desName> -gfile <goalFile> \
      -gset <goalSet> -dir <wdPath> -log <logName>
```

This procedure opens a new Channel and runs the hv program in Slave mode. The new program is running in directory <wdPath>. A new Command Stack is attached to the Channel, and Dut, Goal, and Image Engines are started for the Channel. The new channel uses <logName> if specified. <goalSet> specifies the name of the GoalSet created by sourcing <goalFile> which is to be checked for unreachable goals. <vecName> is the name of a StateVector created by sourcing <initFile>; this StateVector contains the starting state for fixed point computation.

The following procedures are executed in order for the Channel; this is carried out by pushing the commands onto the Command Stack in reverse order. When each task subsequently finishes, the various message handlers pop the next command off the stack and execute it.

- hv_dut_setup – This procedure loads HNL and creates HvDsg in the Slave process.
- hv_load_goals – This procedure causes goals to be loaded into the Slave process.
- hv_start_lmbm – This procedure initiates the fixed point computation in the Slave process.
- hv_finish_lmbm_unreach – This procedure is called when the fixed point computation completes and uses the fixed point to determine which goals are provably unreachable.
- hv_update_goals – Causes unreachable goals to be communicated from the Slave process to the Master process. The Master subsequently communicates the info to other Slave engines which care.

The hv_dut_setup procedure is directly executed and this procedure returns without waiting for anything to finish. TCL knobs used by this procedure are: hvPath, goalFile (if –gfile not used), covVarsFile, workDir (if –dir not used), verbMode.

4.3.5.4. hv_start_lmbm Procedure

This procedure is called to initiate pseudo-LMBM style fixed point computation using symbolic state traversal techniques. This fixed point is subsequently used to prove

unreachable goals. This procedure is called after the DUT and Goals have been properly setup in the Slave process. Procedure prototype is:

```
hv_start_lmbm -chnl <chnlId> -init <initFile> \
      -state <vecName>
```

<chnlId> is the Channel which has had DUT created and Goals loaded properly (as described by previous sections). <vecName> is the name of a StateVector created by sourcing <initFile>; this StateVector contains the starting state for fixed point computation.

Note that current usage does not really do LMBM type fixed point computation because the underlying machine is not partitioned into submachines. If the underlying machine and HvCic were partitioned, the state traversal would be true LMBM. This procedure issues the following commands to the Slave process without waiting for any of them to finish:

- hvflc_create_cover
- hvfml_create_lib
- hvfac_create_rpb_from_hbv_state – The starting state <vecName> is converted to an RPB for use by the Image Computation algorithms.
- hvfmm_compute_lmbm – This procedure actually computes the fixed point. The Image Engine is set to the state "run_lmbm". The resultant fixed point name is "lmbm_fpoint".

The completion of this procedure is detected by receiving a IMG_ENG_LMBM_DONE message from the Image computation engine. A command is popped from the stack when this message is received.

This procedure uses the TCL knobs: verbMode.

4.3.5.5. hv_finish_lmbm_unreach Procedure

This procedure is called after fixed point computation has finished.. The procedure prototype is:

```
hv_finish_lmbm_unreach -chnl <chnlId> \
      -lmbm <lmbmName> -gset <goalSet>
```

The Master process sends commands to the Slave of Channel <chnlId> to cause unreachability computation to occur for the goals in <goalSet> using the fixed point <lmbmName>.

The following commands are sent to the Slave process of <chnlId>:

- hvfac_print_rpb – This is RPB-specific information about the fixed point RPB named "lmbm_fpoint".
- hvfac_count_minterms_in_rpb – This command projects lmbm_fpoint onto the coverage variables contained in InetSet CovVars, then prints some statistics.

- hvgol_report_unreachable – This command makes the fixed point known to the appropriate GoalSet. The number of Reachable, Unreachable, and Unknown goals are printed in a message for consumption by the Master process.

At the completion of this procedure (more specifically, the resultant message from the hvgol_report_unreachable command), the Master process knows how many goals are unreachable, but not which ones. The hv_update_goals procedure describes how this task is completed.

The completion of this procedure is detected by receiving a HVGOL_CMD_POP message with data value "report_unreach". A command is popped from the stack when this message is received.

This procedure uses TCL knobs: verbMode.

4.3.5.6. hv_update_goals Procedure

This procedure is called after a Slave process has determined unreachable goal information. The goals of the Master process must be updated and this procedure initiates the update. Procedure prototype is:

```
hv_update_goals -chnl <chnlId> -gset <goalSet>
```

This procedure sends a hvgol_write_update_info command to the Slave process of Channel <chnlId>. The Goal Engine for that Channel is set to the "update_goal" state and the procedure returns.

The completion of the hvgol_write_update_info command is detected by receiving a HVGOL_GOALSET_WRITE_UPD message. When this message is received, the update file is sourced in the Master process so that goals in the Master are correct. A command is then popped from the command stack and executed.

This procedure uses TCL knobs: verbMode.

4.3.5.7. hv_vcs_initial_run Procedure

This procedure is called when VCS has finished invoking and running its first reset sequence. The procedure prototype is:

```
hv_vcs_initial_run -chnl <chnlId> -gfile <goalFile>
```

This procedure is given the VCS Channel via <chnlId>. <goalFile> is a sequence of calls to the PLI routines $createSignalSetHV, $addToSignalSetHV, and $defineFsmCoverHV which cause creation of goalsets which are compatible with the goalsets created in the formal engines.

The following commands are sent to VCS:

- source <goalFile> – This causes the necessary goals to be created in VCS
- If requested, pre-reached coverage states may be loaded into VCS by calling the $readCoverInfoHV PLI routine. These pre-reached goals are identified by use of the inputVcsRchStates TCL knob.

- The $configStopPointHV PLI routine is called to setup stop conditions for VCS.

- The hv_vcs_stopped command is pushed onto the Command Stack for the VCS Channel. The VCS Engine of the Channel is set to the "running" state.

- A run command is sent to VCS. This is the famous "." command.

The procedure then returns without waiting for completion of the run command. Completion of the initial run is detected by a receiving a VCS_STOP message when vcs is in the "initing" state. Completion of non initial runs is detected by a VCS_STOP message while in the "stop_pending" state.

This procedure uses TCL knobs: verbMode, goalFile, inputVcsRchStates, initCycles, initRunsCnt.

4.3.5.8. hv_vcs_stopped Procedure

This procedure is called when VCS has stopped after running biased random simulation. This procedure examines various pieces of information to determine what should be done next. Procedure prototype is:

```
hv_vcs_stopped -chnl <chnlId>
```

VCS has stopped running on <chnlId>. The following decision process takes place, depending on CTG analysis phase:

- MODE_INIT – This is the mode where CTG is reaching as many goals as reasonably possible from the reset state using VCS. If the number of VCS runs from the reset states is less than initRunsMax, or if formal reachability is inhibited (using inhibitFormalReach), then 1) tell VCS to perform a reset, 2) push hv_vcs_stopped onto the VCS command stack, 3) initiate a VCS run, 4) return.

  If the number of initial random runs is greater than initRunsMax then a formal search is initiated from the reset state. CTG analysis phase is set to MODE_INIT_SAT (if satCycMax is > 0), else phase is set to MODE_INIT_SYM). Following this phase adjustment, the following steps are taken: 1) tell VCS to perform a reset then stop in the reset state, 2) push hv_run_formal onto the VCS command stack, 3) tell VCS to run, 4) tell VCS to write state to a file using $reportDutStateHV, 5) tell VCS to write newly-reached goals to file using $reportNewCoverDataHV, 6) return.

  An additional decision process in this mode is to choose when unreachability analysis is initiated. Unreachability is launched and runs concurrently with reachability. Unreachability is launched from the same reset state as the first formal search, or earlier if unreachInitCnt is non-0. This knob requests that the unreachInitCnt-th reset state will be used to launch unreachability. If unreachability is launched, then the hv_do_lmbm_unreach procedure is called and VCS is told to write state into the file 'lmbmInit.tcl' using the $reportDutStateHV PLI routine. The VCS engine state is set to 'reach_lmbm_after_reset' in order for the message handler to track what is going on.

- **MODE_INIT_SAT, MODE_INIT_SYM** – This is the phase where CTG is reaching as many goals as possible using SAT/SymbolicSim from the reset state. VCS has stopped at some non-reset state and a formal search is to be setup and kicked off from the reset state. The following actions are taken: 1) tell VCS to perform a reset then stop in reset state, 2) push hv_run_formal onto the VCS command stack, 3) tell VCS to run, 4) tell VCS to write state to a file using $reportDutStateHV, 5) tell VCS to write newly-reached goals to file using $reportNewCoverDataHV, 6) return.

- **MODE_SAT, MODE_SYMSIM_SAT, MODE_SYMSIM** – In these phases, CTG is using formal searches from deep states reached using VCS. At this point, VCS has stopped at one of these deep states and this routine will either kick off a formal search, or reset VCS and try to reach more interesting deep states. The following actions are taken (in pseudo-code form):

```
If ((stopped due to cycleLimit) &&
   (cycleLimitMax <= cycleLimitStopCnt))
   push hv_vcs_stopped onto the vcs command stack
   tell VCS to reset
   tell VCS to run
else
   if (phase is MODE_SAT)
         push hv_run_formal -type SAT on VCS stack
   else if (phase is MODE_SYMSIM)
         push hv_run_formal -type SYM on VCS stack
   else
         if (last formal was SAT)
               push hv_run_formal -type SYM
         else
               push hv_run_formal -type SAT
         endif
   endif
   tell VCS to dump state ($reportDutStateHV)
   tell VCS to dump new goals ($reportNewCoverDataHV)
endif
```

- **MODE_DONE** – VCS has reached all requested goals, and CTG needs to update the goals in the Master process then halt cleanly. The following actions are taken: 1) push hv_run_formal onto the VCS command stack, 2) tell VCS to dump state using $reportDutStateHV, tell VCS to dump all recently reached goals using $reportNewCoverDataHV.

When the $reportNewCoverDataHV PLI routine is issued, completion is detected by receipt of a VCS_NEW_WRITTEN message from VCS. A command is popped from the stack in response to this message.

This procedure uses TCL knobs: verbMode, initRunsMax, inhibitFormalReach, updateFile, stateFile, resetConstFile, runConstFile, satCycMax, unreach, stateVector, goalFile, workDir, design, goalSetNames, cycleLimitMissMax.

4.3.5.9. hv_vcs_replay_done Procedure

This procedure is called when VCS has completed replaying a trace generated from a formal search engine. Procedure prototype is:

```
hv_vcs_replay_done -chnl <chnlId> -action <actType>
```

<actType> specifies which action the initiator of the replay thought might be appropriate; must be "RUN" or "BACK2BACK".

- **RUN** – VCS is to run for a period of time using biased random simulation. The hv_vcs_stopped procedure is pushed onto the VCS command stack, and VCS is then told to run.

- **BACK2BACK** – This action is to use "Back-to-Back" formal searches. In this case, random simulation is not allowed to run immediately after a formal search has succeeded in hitting a new goal. hv_run_formal is pushed onto the VCS command stack. VCS is told to 1) write its state to file by $reportDutStateHV, then 2) write recently reached goals to file by calling $reportNewCoverDataHV

This procedure uses the TCL knobs: verbMode, updateFile, stateFile, resetConstFile.

4.3.5.10. hv_start_symsim_mgr Procedure

This procedure is called to setup the Symbolic Simulation manager prior to performing any searches using that engine. This procedure is called once at the beginning of a session, after goal and dut engines are properly set up, Ord, Sth, and Mch are already built, etc. Typically called from hv_setup_formal. Prototype is:

```
hv_start_symsim_mgr -chnl <chnlId>
```

<chnlId> is the channel that should have a Symbolic Simulation manager built. The hvprm_sim_manager create command is called in the child process. It is expected that hvmch_$design, hvflc_$design, and hvord_$design are valid Mch, Flc, and Ord objects, respectively.

Completion of this sequence is detected by receipt of HVSYM_MGR_SETUP message. A command is popped from the command stack in response to this message.

4.3.5.11. hv_run_formal Procedure

This procedure is called to initiate a formal search on a properly opened and set up Channel. Function prototype is:

```
hv_run_formal -chnl <chnlId> -sfile <startFile> \
      -update <updateFile> -const <constFile>
      -type <rchType>
```

<rchType> must be "SAT" or "SYMSIM", depending on which engine should be used for the formal search. <startFile> is a TCL file to be sourced in the Slave process to create a StateVector; this StateVector defines the starting state for the search engine. <updateFile> is a TCL file to be sourced in the Slave and Master processes to update the GoalMgr with states reached by VCS since the last <updateFile> was generated. <constFile> is a TCL file to be sourced in the Slave process to cause the search engine to see certain primary inputs as of a constant value. Note: This <constFile> is expected to be obsoleted when HNL-based environmental modeling can specify the same data.

Actions carried out in this procedure are:

1. Issue commands to all Slave processes running symsim or sat engines to "source <updateFile>". This causes all processes to have full knowledge of what goals have been reached by VCS.

2. Push hv_formal_stopped onto the Command Stack of the channel to perform the formal search.

3. If we are initiating back-to-back searches, adjust resource limits accordingly. If <rchType> is "SAT, iteration limit is set to satB2BCycMax or satCycMax. If <rchType> is "SYMSIM", iteration limit is set to symsimB2BCycMax or symsimCycMax.

4. If <rchType> is SYMSIM, source <constFile> in the child process, then source $SYNOPSYS/auxx/ctg/tcl/runPrm.tcl in the Slave. If <rchType> is SAT, then call the hvcnf_sat command in the child process.

5. The procedure then returns.

Completion of the formal search is detected by receipt of either HVSYM_SIM_DONE or HVSAT_REACH_DONE messages. A command is popped from the command stack in response to either of these messages.

This procedure uses the following TCL knobs: symsimCpuMax, symsimCycMax, symsimB2BCycMax, satCpuMax, satCycMax, satB2BCycMax, goalSetNames,

4.3.5.12. hv_formal_stopped Procedure

This procedure is called when a formal search engine has terminated a search. The search may have terminated after finding one or more goals, or it may have terminated after exceeding some user-provided resource limits. Procedure prototype is:

```
hv_formal_stopped -chnl <chnlId> -type <rchType>
```

<rchType> is either "SAT" or "SYMSIM" and is the type of search engine which was initiated on the Channel of <chnlId>.

Note that when the search engine completed, it reported how many goals were found; this data recorded in either satHandler.tcl or symsimHandler.tcl.

The decision process of this routine is outlined by the following pseudo-code.

```
If (phase is MODE_INIT_SYM)
      If (search found 0 goals)
            If (satCycMax>0)
                  Set phase to MODE_SAT
            Else
                  Set phase to MODE_SYMSIM
            Endif
            Set non-init-phase VCS stop conditions \
                  & freshness using $configStopPointHV
            push hv_vcs_stopped on VCS cmd stack
            set VCS engine state to "running"
            tell VCS to run
      Else
            Push hv_vcs_replay_done on VCS cmd stack
            Tell VCS to bias weights
```

```
            Tell VCS to replay a trace
            Set VCS engine state to "running"
            Tell VCS to run (do the replay)
      Endif
Elseif (phase is MODE_INIT_SAT)
      If (search found 0 goals)
            If (symsimCycMax>0)
                  set phase MODE_INIT_SYM
            Else
                  set phase MODE_SAT
            Endif
            If (phase is MODE_INIT_SYM)
                  Call hv_run_formal
            Else
            Set non-init-phase VCS stop conditions \
                  & freshness using $configStopPointHV
            push hv_vcs_stopped on VCS cmd stack
            set VCS engine state to "running"
            tell VCS to run
      Else
            Push hv_vcs_replay_done on VCS cmd stack
            Tell VCS to bias weights
            Tell VCS to replay a trace
            Set VCS engine state to "running"
            Tell VCS to run (do the replay)
      Endif
Elseif (phase is MODE_SAT, MODE_SYMSIM_SAT, MODE_SYMSIM)
      If (search found 0 goals)
            If (phase is MODE_SAT)
                  If (symsimCycMax>0)
                        set phase MODE_SYMSIM_SAT
                  Endif
            Elseif (phase is MODE_SYMSIM_SAT)
                  If (lastFormalEng was SAT)
                        Set phase to MODE_SYMSIM
            Endif
            If (#consecMisses > formalMissThresh)
                  Push hv_vcs_stopped on VCS stack
                  Set #consecMisses to 0
                  Tell VCS to reset
            Elseif (lastFormalEng was SYMSIM)
                  Push hv_vcs_replay_done on VCS stack
                  Tell VCS not to bias weights
                  Tell VCS to replay a trace
            Else
                  Push hv_vcs_stopped on VCS stack
            Endif
      Else
            If (#consecHits > formalClearThresh)
                  Tell VCS to clear bias weights
            Endif
            If (back2back enabled)
                  Push hv_vcs_replay_done on VCS stack \
                        (with BACK2BACK action)
            Else
                  Push hv_vcs_replay_done on VCS stack \
```

```
                        (with RUN action)
            Endif
            Tell VCS to bias weights
            Tell VCS to replay
        Endif
        Set VCS engine state to "running"
        Tell VCS to run
Endif
```

This procedure uses the following TCL knobs: verbMode, satCycMax, symsimCycMax, freshStartFrac, freshSampFrac, rsimCycles, freshSuppress, updateFile, stateFile, resetConstFile, formalMissThresh, formalClearThresh, backToBackEnabled.

4.3.5.13. hv_update_cover_metric Procedure

This procedure computes the actual cover metric used to determine progress made in the overall coverage session. This is a function of all goals/goalsets currently defined. Sets the covMetric TCL variable. Procedure prototype is:

```
hv_update_cover_metric
```

This procedure takes no arguments.

4.3.6. TCL Knobs and Variables

The CTG control mechanisms, heuristics, and decision algorithms are largely implemented in TCL code running in the Master process. This TCL code allows for many user-specifiable configuration settings; these are called TCL "knobs". This control TCL code also needs to keep track of session progress and status; this makes use of TCL "variables" which change as the session progresses.

The TCL language is limited in variable scope and data types. Variables are visible either globally or local to a single procedure. Global variables used in a procedure must be declared in that procedure as 'global' before use in the procedure. TCL variables must be of type string or array; array variables are indexed by strings and are one-dimensional.

All TCL knobs and variables are organized as elements of a single global array variable named "ctgInfo". The previously mentioned knob "satCycMax", for example, is actually referenced as "ctgInfo(satCycMax)". When a TCL knob or global variable is added to the ctgInfo array, a documentation entry is also entered in a corresponding array "ctgInfoDoc". The documentation line for ctgInfo(satCycMax) is thus contained in ctgInfoDoc(satCycMax) and its value is: "Maximum SAT cycles allowed per search". A TCL procedure hv_print_ctg_knobs is provided to print knob values and documentation; this procedure was described earlier.

This remainder of this section documents knobs and variables in use at the time of writing this document.

4.3.6.1. TCL Knobs

The following knobs are used to configure the CTG TCL control code. In each case, the knob name is provided without the "ctgInfo()" for clarity. Knob default values are also provided, where relevant.

- **workDir** "." - Directory that child process are started in, by default.

- **stateFile** "stateFile.tcl" - Name of temporary file VCS uses to write current design state into. When sourced, a StateVector is created then each state element is set to the current value (0/1) in the VCS simulation session.

- **stateVector** "StateVec" - Name of State Vector that updateFile.tcl saves state in when it is sourced

- **updateFile** "updateFile.tcl" - Name of temporary file VCS uses to store reached states in when this info is shared with other processes. When sourced, all goals and goalsets are updated with goals which have been reached since the last time VCS was asked to report newly reached states.

- **runConstFile** "<no_consts>" - Name of file which can be sourced to set certain signals to constant values suring symbolic simulation. This file consists of a series of hvprm_input_set_to_constant commands

- **resetConstFile** "<no_consts>" - Set certain signals to constant values for Symbolic Simulation from the reset state. Similar to runConstFile, except used only when simulating from reset state.

- **goalFile** "goalFile" - Basename of files used to create goals in hv and VCS child processes. It is expected that <goalFile>.vlog can be sourced in VCS, and <goalFile>.tcl can be sourced in hv.

- **goalSetNames** "<no_goalsets>" - Space-delimited list of GoalSet names which are created by sourcing the above <goalFile>.

- **goalNames** "<no_goals>" - Space-delimited list of Goal names which are created by sourcing the above <goalFile>.

- **covVarsFile** "covVars.tcl" - Name of file to source in hv which creates an InetSet named 'CovVars'. This InetSet contains all coverage variables used in all goals and goalsets analyzed in this session.

- **unreach** "UNREACH_NONE" - Knob which specifies style of unreachability analysis to perform. Valid values are:

    - UNREACH_NONE: No unreachability used

    - UNREACH_LMBM: Use lmbm-type fixed point computation

- **unreachInitCnt** 3 - Specify which reset state should be used to initiate unreachability from. In practice, not all verilog registers in a design are set to a specific value by the reset sequence. Thus, sometimes X logic values may take many simulation cycles and/or resets to flush from the system. In these cases, the first reset state generated may not be a good start-point for unreachability analysis. This knob specifies that the i-th reset state should be used for unreachability computations. Unreachability will start no later than when the first formal search is initiated. I.E. when MODE_INIT is exited.

- **inhibitFormalReach** 0 - If this knob is non-0, then formal search mechanisms are not used. CTG stays in MODE_INIT forever.

- **verbMode** 1 - Specifies verbosity mode to use in TCL control code running in the Master process. Set to one of:
  - ctgInfo(verbUSER): Not much verbosity
  - ctgInfo(verbAPPLICATION): a little more
  - ctgInfo(verbDEVELOPMENT): still more
  - ctgInfoDoc(verbPANIC): so much we get lost
- **freshMode** "START_FRESH_OFF" - This knob specifies when freshness detection should start to be used. Valid values are:
  - START_FRESH_OFF: dont use freshness.
  - START_FRESH_DYNAMIC: use freshness after the first miss
  - START_FRESH_ALWAYS: freshness on from the very start
- **freshSuppress** 3000 - Number of cycles to suppress freshness detection at the beginning of a random simulation run
- **freshSampFrac** 1000 - Sample fraction, in 1/1000 percent units, to use for freshness detection. If a given state has been seen less than this percentage, then that state is said to be "fresh".
- **freshStartFrac** 10000 - Start fraction used for freshness detection. When VCS is asked to save current state to a file, this is taken to be a start point for a formal search. If a given state has been used as start point less than this percentage of all times a start has been initiated, then the state is said to be fresh.
- **cicBlkSize** 50 - The size of blocks to use for HvCic object used for unreachability analysis.
- **cicMaxLimit** 3000 - The maximum number of latches to use in the HvCic object used for unreachability analysis.
- **rsimCycles** 10000 - The number of clock cycles VCS will perform random simulation before stopping for direction. VCS runs this many clock cycles past the time the last new state (i.e. goal) was reached. The TCL control code in the Master process then determines if the current state should be used for formal search, if a reset should occur, etc.
- **initCycles** 2000 - The number of clock cycles of random simulation to perform after a reset sequence. VCS will run this many cycles past the last new state (i.e. goal) seen
- **initRunsMax** 2 - How many sequences of Reset+RandomSim will be performed before CTG exits MODE_INIT

- **dynamicBiasCycles** 1000 - How many cycles of dynamically biased random simulation will be run after a successful formal search to hit a new goal. After this many cycles, user-biased random simulation will be used.

- **satCpuMax** 600 - Maximum number of CPU seconds which is allowed for a formal search using SAT.

- **satCycMax** 50 - Maximum number of simulation steps (i.e. clocks) which are allowed for a SAT formal seach

- **satB2BCycMax** 10 - Maximum number of simulation steps allowed for a SAT search when used as the second search in a BackToBack search. Generally this number is less than satCycMax.

- **symsimCpuMax** 600 - Maximum number of CPU seconds which is allowed for a formal search using Symbolic Simulation.

- **symsimCycMax** 50 - Maximum number of simulation steps (i.e. clocks) which are allowed for a Symbolic Simulation formal seach

- **symsimB2BCycMax** 25 - Maximum number of simulation steps allowed for a Symbolic Simulation search when used as the second search in a BackToBack search. Generally this number is less than symsimCycMax.

- **formalReplayFile** "prm_sequence" - Name of temporary file used by formal search engines to save replay traces in. This file is then read by VCS to replay the trace

- **cycleLimitMissMax** 2 - This knob specifies how many times a formal search may be initiated from a non-fresh state when a miss occurs. The idea is that often a search from non-fresh state has no hope of succeeding. Note that these seaches occur when rsimCycles of simulation have occurred in VCS since that last goal was hit.

- **backToBackEnabled** "OFF" - This knob specifies if BackToBack should be enabled or not. Values of OFF and ON are allowed. BackToBack causes a formal search to be performed immediately after a formal search has succeeded in hitting a new goal. The default behavior is to perform dynamically biased random simulation from these points.

- **formalMissThresh** 2 - Number of consecutive formal searches which may fail before VCS is instructed to perform a reset.

- **formalClearThresh** 3 - Number of consecutive formal searches which may succeed before dynamic biasing weights are cleared in VCS.

- **vcsTestFile** "<no_testfile>" - Name of file which the generated test is to be saved in. This test may be used later as a stand-alone regression test in VCS. Two files are saved:

    - <testFile>.ctg contains the control program to reproduce the input stimulus.

    - <testFile>.vgol specifies the goals and goalsets used for the test.

- **inputVcsRchStates** "<no_inputfile>" - This file specifies goals which are considered to have been previously reached. These goals are classified as "reached" at the beginning of the CTG session, and no attempt is made to generate a trace to these goals.

- **outputVcsRchStates** "" - This file specifies a file to record goals which are reached during this analysis session. This file format is suitable as input to later CTG sessions.

4.3.6.2. Variable Usage

This section lists glaobal variables used by multiple TCL procedures in the TCL control code running in the Master process.

- **ctgMode** - This variable contains which phase CTG is currently in. One of MODE_INIT, MODE_INIT_SAT, MODE_INIT_SYM, MODE_SAT, MODE_SYMSIM_SAT, MODE_SYMSIM, or MODE_DONE

- **unreachStarted** - Variable that says unreachability has been started (if non-0)

- **initRunsCnt** - Number of MODE_INIT simulation runs which have been performed from the reset state.

- **covMetric** - This variable is the number of goals left to be classified

- **formalMissCnt** - Number of consecutive formal search misses seen.

- **formalHitCnt** - Number of consecutive formal search hits seen.

- **formalTraceCnt** - How many goals the last formal search found

- **lastReachType** - Which type formal search engine was used last

- **back2BackInProcess** - A BackToBack search is in progress

- **lastVcsStopReason** "<no_reason>" - Reason that the lst VCS simulation stopped. One of VCS_STOP_UNKNOWN, VCS_STOP_DONE, VCS_STOP_CYCLE_LIMIT, VCS_STOP_FRESH, VCS_STOP_END_REPLAY

- **cycleLimitMissCnt** - How many formal search misses have when seen when at cycle limit stop point.

- **hvPath** "/vobs/propver/src/hv/bin-gccsparcOS5/hv-g" - Pathname to where hv executable lives

4.3.6.3. ctgVcsCoverObjs Variable

When a coverage object is created in VCS(e.g. creating a GoalSet by using the $defineFsmCoverHV PLI routine), it is given a unique "handle". This handle is then used as input to other PLI routines during the CTG session. The same coverage object is known by name in the Master process and other hv Slave process. The ctgVcsCoverObjs variable is an array that maps a coverage object name (e.g. the GoalSet name) to its

corresponding handle in the VCS process. For example: ctgVcsCoverObjs(GoalSet1) = 1.

4.3.6.4. ctgVcsCoverStatus Variable

The ctgVcsCoverStatus variable is an array which records current classification counts for the goals which correspond to a coverage object in VCS. Index into the array is the VCS handle of the object. Data value is a string of 3 space-delimited numbers: ctgVcsCoverStatus(1) = "#unknown #unreachable #reached".

5. Goal Creation and Usage

The creation, maintenance, and use of goals is central to the CTG methodology. Goals are, after all, what are being "covered", "reached", "proven", "classified", etc. The hv program provides a number of goal-related TCL commands which are documented in this section. In addition, the PLI code linked into VCS provides certain goal-related capabilities which are discussed.

5.1. Introduction

In general, each of the Slave processes in use in the CTG system will need some or all of the goal information collected by the other Slave processes. The TCL control code running in the Master process is responsible for updating the necessary goal information in a Slave process before asking for an analysis by that Slave.

Note that in many cases a Slave does not need all accumulated goal information before carrying out its portion of the CTG session. For example, unreachability analysis performed by LMBM fixed point computation has no need to know that some goals have already been reached.

At all times, the Master process is the central keeper of the union of all goal results determined by all child processes. The Master process TCL code is responsible for propagating all generated results from Slaves to the Master, and from the Master to Slaves that need the information.

5.2. HvGol Package TCL Commands

The HvGol package provides the following TCL commands which are used to manage goal information within the CTG system. Note that not all hvgol_* commands are used currently by CTG. Only those commands in use or otherwise of interest are mentioned here.

- hvgol_create_goal - This command creates a goal from a single signal or an HbvStateVector. The goal is given a name at creation time.

- hvgol_create_goalset - This command creates a goalset given an InetSet which contains the coverage variables of interest. Given N coverage signals which may take binary logic value B={0,1}, then 2**N goals are implied by the GoalSet and have all logic values given as B**N.

- hvgol_goalset_add_info – This command is used to incrementally specify reached/unreachable state information for an existing GoalSet. Unreachable states must be represented as either fixed point HvRpb or as a HvHbv StateVector. Reached states are specified as HvHbv StateVector .

- hvgol_goalset_relations - This command copies out some or all of the HvSth_Relation_t's which are used internally to represent the Unknown, Unreachable, and Reached goals of the GoalSet. The user provides the names used to create named HvMgr instances of type HvSth_Relation_t.

- hvgol_write_update_info – This command writes into a file the HvSth_Relation_t objects which represent the unknown, unreachable, and reached relations for a given GoalSet. This information is suitable for reading in using the hvgol_read_update_info command.

- hvgol_read_update_info - This command reads goalset information written by one hv session, and used in another. The provided reached, unreachable, and unknown Relations from one goalset are used to update the same GoalSet with new information.

- hvgol_report_goal - This command reports various information about a Goal.

- hvgol_report_goalset - This command reports various information about a GoalSet

- hvgol_report_unreachable – This command reports reached/unreachable state information to a goal or goalset. This command is somewhat mis-named, as the semantics are that the caller is reporting to the Goal, rather than vice-versa. Unreachable states are in RPB fixed point form. Unreachable states are specified as an HvHbv StateVector.

5.3. HvPli Services For VCS Usage

Certain goal capabilities are implemented in the VCS Slave process using standard PLI capabilities. VCS is invoked in interactive mode so that these PLI routines may be "called" by the Master process. The PLI routines provided for goal creation/maintenance are the following:

5.3.1. createSignalSetHV PLI Routine

```
$createSignalSetHV("<setName>")
```

This PLI routine is used to define a SignalSet in the VCS process. A SignalSet is entirely analogous to an InetSet in one of the hv proceses. The "setName" argument is a string that is expected to correspond to a InetSet in the Master process.

5.3.2. addToSignalSetHV PLI Routine

```
$addToSignalSetHV("<setName>", <signal>, "<addMode>")
```

This PLI routine is used to add a Verilog signal to the named SignalSet. <signal> may be either a net or register in verilog. <addMode> is a string that is either PRE or POST to indicate the signal should be prepended or appended to the SignalSet. The order of the signal in the SignalSet is expected to be the same as the order of signals in the corresponding InetSet.

5.3.3. reportSignalSetHV PLI Routine

```
$reportSignalSetHV("<setName>","<fileName>","<fmt>")
```

This PLI routine writes a SignalSet given by <setName> out to <fileName> in a specified <format>. <format> is a string that is either "HV" or "PLI". These formats are suitable to be sourced in either hv or VCS programs to reconstruct the SignalSet.

5.3.4. defineFsmCoverHV PLI Routine

```
$defineFsmCoverHV("<objName>", "<signalSetName>",\
        <sysClk>, <offSet>, <fsmHandle>)
```

This PLI routine creates an Fsm type coverage object named <objName>. The goals are defined as all [1,0] combinations of the signals contained in <signalSetName>. If there are N signals in SignalSet, then 2**N individual goals are defined with this command. These signals are sampled relative to the positive edge of the clock signal <sysClk>. <offSet> specifies a non-negative offset by which the sample time can be adjusted so that the sample happens a short time after the positive edge of <sysClk>. This routine allocates a unique positive integer handle for the coverage object; this is returned as <fsmHandle>.

5.3.5. defineBitToggleCoverHV PLI Routine

```
$defineBitToggleCoverHV("<objName>", "<signalSetName>",\
        <sysClk>, <offSet>, <fsmHandle>)
```

This PLI routine creates a Toggle coverage object named <objName>. Signals given by the <signalSetName> are to be toggled to both 1 and 0; these are the goals. If there are N signals in SignalSet, then 2*N individual goals are defined with this command. These signals are sampled relative to the positive edge of the clock signal <sysClk>. <offSet> specifies a non-negative offset by which the sample time can be adjusted so that the sample happens a short time after the positive edge of <sysClk>. This routine allocates a unique positive integer handle for the coverage object; this is returned as <fsmHandle>.

5.3.6. getCoverMetricHV PLI Routine

```
$getCoverMetricHV(<covHandle>, <covNumber>)
```

This routine determines how many goals remain unclassified in the coverage object identified with <covHandle>. If <covHandle> is –1, then all currently-defined coverage objects are used and the returned <covNmuber> is the summation of all unclassified goals.

5.3.7. reportNewCoverDataHV PLI Routine

```
$reportNewCoverDataHV(<covHandle>, "<fileName>")
```

This PLI routine writes to <fileName> all goals which have been covered since the last call to this routine. <fileName> must be writeable, and will be overwritten by this routine. The output file is expected to be sourced by the hv program to update the corresponding GoalSets in the hv process. File contents are the following:

- create a StateVector with the same name as the GoalSet (hvhbv_new_vector –init DC ...)
- Set each care bit in the StateVector using hvhbv_set_vector_bit
- Update the GoalSet using hvgol_goalset_add_info to present the properly loaded StateVector as a new reached state for the GoalSet.

5.3.8. informUnreachCountHV PLI Routine

```
$informUnreachCountHV(<covHandle>, <#unreach>)
```

This PLI routine is used to inform the PLI code that a certain number of states of the given coverage object have been proven unreachable. The effect of this routine is to adjust the coverage metric for the coverage object given by the handle <covHandle>. Note that this command is additive for a given coverage object; this allows for unreachable goals to be proven throughout an analysis session by different unreachability algorithms.

5.3.9. printCoverInfoHV PLI Routine

```
$printCoverInfoHV(<covHandle>,"<fileName>","<fmt>")
```

This PLI routine writes out reached state coverage information out to the file named <fileName>; this file must be writeable and will be overwritten. If <covHandle> is –1, then all current coverage objects are written, else the one specified by the handle <covHandle>. <format> is a string and is one of "COMPACT", "COMPACT_REACH", or "HUMAN". "COMPACT" and "COMPACT_REACH" formats are intended to be readable using the $readCoverInfoHV PLI routine. "COMPACT" simply writes out the cover object name and coverage signals. COMPACT_REACH writes cover object name, coverage signals, and reached states. "HUMAN" writes information out in a human-readable form.

5.3.10. readCoverInfoHV PLI Routine

```
$readCoverInfoHV("<fileName>", <use_rch>)
```

This PLI routine reads coverage object definitions from the file given by <fileName>. <use_rch> is an integer; if non-0, then any reached states contained in <fileName> are taken to be previously-reached states and are immediately considered as reached. The next call to $reportNewCoverDataHV will report these states as reached.

5.4. Usage Scenarios

When goals are determined to either be unreachable or reached, this information must be communicated to other parts of CTG which need this information. This section describes these system-level interactions.

5.4.1. Unreachable State Computation

When unreachability analysis has been completed by one of the Slave processes, the unreachable state data must be reported to the Master process, and conveyed to the VCS process so that the coverage metric can be adjusted properly. The Slave process is told to write unreachable states to file using the hvgol_write_update_info. The Master process then uses the hvgol_read_update_info command to update its central copy of the GoalSet information. The number of unreachable states is reported to the VCS Slave by calling the $informUnreachCountHV PLI routine.

If a SAT solver is active, then the unreachable states are also read into that process.

5.4.2. Reached State Information

The CTG system does not consider a Goal to have been reached until VCS has generated an input sequence that actually reaches a state which covers the Goal. When a Goal has

been reached, then this information must be conveyed to the Master process, as well as any Slave processes which can take advantage of this information.

This update process occurs each time VCS stops simulation and waits for input directive from the Master process. The following actions then occur:

- VCS is told to write out newly reached goals using the $reportNewCoverDataHV PLI routine.
- The Master process updates its copy of the GoalSet by sourcing the file of update information. As previously described, this file creates a vector of Don't-Care bits then sets all care bits to the appropriate value. This StateVector is then used with the hvgol_goalset_add_info command.
- All Slaves used for reachability analysis are also told to source the update file.

6. UI Interface

The User Interface of Figure 2 interacts with the core CTG system via the Master process. As previously described, commands are issued to the Master process via a unix pipe which feeds stdin of the Master process. All native hv commands, as well as the externally usable TCL control routines in hvRecipe.tcl, are available for UI needs. This interface allows the UI to be written in either Graphic or Textual modes without any changes, from the Master process perspective. Refer to some other UI-specific document for detailed issues about User-level interactions with CTG.

7. VCS Interface

As indicated earlier, the explicit state simulation engine integrated into CTG is based on VCS. CTG capabilities are linked into VCS directly via PLI techniques for direct simulation access, and using VERA mechanisms for testbench control, constraints, and biasing.

For detailed descriptions of VERA coding and compilation issues for use with CTG, refer to appropriate CTG VERA documentation. This section describes the specific control & interface mechanisms for CTG interactions with VERA code.

For more detailed VCS and PLI descriptions, refer to appropriate product literature. This section document CTG interfacing and control mechanisms present in the VCS Slave process.

7.1. PLI Interfacing – hvPli

As part of a product deliverable, CTG provides PLI routines embodied in a precompiled hvPli.a archive library which is linked to the VCS compiled simulator. A hvPli.tab interface file is provided so that these PLI routines can be called from the user PLI code, as well as interactively from the VCS interactive command line.

PLI routines used in the CTG product are the following. Note that Goal-specific PLI routines were described earlier in this document and are not repeated here.

7.1.1. startHV PLI Routine

```
$startHV(<rootInst>, "<mode>", "<regMapFile>",\
      "<inGoalFile>","<outGoalFile>")
```

This PLI routine is called at the very start of a CTG session to properly setup and initialize the CTG PLI mechanisms. This routine is called either by the Master process when creating a testcase, or by the CoverBooster program when replaying a previously-created testcase.

<rootInst> is a module in the verilog netlist which is being formally analyzed.

<mode> specifies whether a test is being created ("SLAVE_CREATE") or a previously-created test is being replayed ("FREE_REPLAY").

<regMapFile> is a file which specifies the HNL name of sequential elements in the formally-analyzed design. These names are decoded, together with <rootInst>, to identify registers in the verilog which comprise "state" of the Design Under Test. This information is especially relevant to the $reportDutStateHV PLI routine. Only registers which are relevant to the HNL model (i.e. those verilog registers "inferenced" as sequential elements by the synthesis process) are reported as DUT state for use as start point of formal searches. <regMapFile> is required for mode "SLAVE_CREATE" and is not required for "FREE_REPLAY".

<goalFile> specifies SignalSet and coverage objects. This file may be omitted, but no coverage can be measured until cover objects are defined. As discussed earlier, other PLI routines are provided to define coverage objects after this point.

<outGoalFile> is optional and is used to specify a file which reached goals will be saved in upon VCS termination. This format is suitable to be used as input <goalFile> or as input to the $readCoverInfoHV routine.

7.1.2. reportDutStateHV PLI Routine

```
$reportDutStateHV("<outFile>", "<Xmode>");
```

This routine writes the current state of the DUT out to file <outFile>. The file format is intended to be sourced by the hv program; a HvHbv StateVector named "StateVec" is created for the machine named "hvmch_$design". Each bit of StateVec is assigned the current logic value of the corresponding verilog register. Only registers indicated by the <mapFile> of $startHV are assigned values. The <Xmode> parameter is a string which indicates what logic value should be used for registers which currently have the logic X value. If <Xmode> is "DC", then those X-ed registers have Don't Care value; if <Xmode> is "CONST", then those X-ed registers will be assigned logic 0 value.

When a register is encountered in the X state, a message is printed (maximum of one message). A comment is written at the end of <outFile> which documents how many registers were in the X state.

7.1.3. informSearchResultsHV PLI Routine

```
$informSearchResultsHV(<#reached_goals>)
```

This PLI routine is used to tell the VCS-resident hv code that the previous search was either successful or not. This information is used to determine applicability of a particular state for use as a future search start point. For example, if a particular state was used as the start state for a formal search, and that search failed after 2 hours, then the state might not be a good point to stop and search from in the future.

The <#reachedGoals> is how many goals were reached by a formal search engine from the state last used by the last call to $reportDutStateHV.

7.1.4. configStopPointHV PLI Routine

```
$configStopPointH(<trigReg>, <covHandle>, \
   <samplePercent>, <startPercent>, <cycCnt>, <suppCnt>)
```

This PLI routine is used to tell the VCS-resident HV a simulatiton run should pause and allow control decisions to be considered by the Master process.

<trigReg> is a verilog wire which will be toggled when a "good" stop point has been reached. If <trigReg> is null, then the PLI routine tf_dostop() is called.

<covHandle> specifies the coverage object to which this stop configuration applies. If <covHandle> is –1, then this information applies to all current coverage objects.

<samplePercent> is an integer, in 1/1000 percent units, which specifies an upper bound for how often a state can be "visited" and still be considered "fresh". For example, if a coverage object has been sampled 1000 times and has been in a given state S 10 of those 1000 times, then S has been visited 10/1000 or 1% of the total sample times. If <samplePercent> is greater than 10, then S is considered fresh and may cause a simulation stop to occur. Note that "state" consists of only the coverage variables used by the coverage object; not all registers of the design.

<startPercent> is an integer, in 1/1000 percent units, which specifies an upper bound for how often a state may be used as a "start state" for formal searches. Each state at the time $reportDutStateHV is called is considered a "start state" for a formal search.

Note: it is acknowledged that this is not quite always true. However, in the current system, this is very nearly a true statement.

As an example, if a state S has been used for start state 3 times, and 30 formal searches have been performed, then <startPercent> must be greater than 100 (10%) in order for S to be considered "fresh". A special case happens if a previous search from state S resulted in no goal reached; in this case, S will never be used again as a stop point.

<cycCnt> specifies a maximum number samples which are allowed to pass until a stop is called. This limit is relative to the last goal reached in the coverage object, or relative to the last reset sequence being executed. If <cycCnt> is 0, then a stop will only occur for fresh states.

<suppCnt> is a freshness "suppress" count. Freshness detection is suppressed for this many samples after a new goal is reached.

Note that if either percent is 0, then that metric is turned off for freshness detection. If both percentages are non-0, then a trigger (i.e. stop) condition is the AND of both percentages.

7.2. Verilog TestBench Top Module

The verilog testbench top file is generated by the vera synthesis tool. This "top.v" file instantiates the verilog module being verified, as well as VERA constraint and biasing mechanisms. Refer to appropriate documentation for more details on this process. This section discusses control and interface aspects contain in the top module.

Verilog Control and Status Variables

As outlined earlier in this document, the TCL control code running in the Master process needs to tell the VCS Slave to do a number of things. For example, "execute a reset

sequence", "replay a generated trace", "clear dynamically-biased weights", etc. The mechanisms used to convey these commands are to 1) call various CTG-supplied PLI routines, or 2) set various control flags in the verilog testbench top. These control flags are periodically checked by the VERA VCS code which controls the VCS Slave process. This VERA control code is called "CoverBooster" and is described in a later section of this document.

Flags are defined as verilog "reg"s, and are set to 1 or 0 interactively by using the native VCS command "set". For example, given a statement:

```
reg foo;
```

Then the following interactive VCS command will set foo to 1.

```
cli_1> set foo=1
```

Control flags which are used in the top module of the verilog testbench are the following:

- hvResetFlag – This flag is used to tell CoverBooster to apply the reset sequence at the next possible time. This calls the reset() task in the design_interface implemented in VERA code. This flag is initialized to 1 so that a reset is performed as the very first action.

- hvReplayTraceFlag – This flag tells CoverBooster to replay a replay trace generated by one of the formal search engines. The filename used to contain the trace is defined historically as "prm_sequence". This flag is initialized to 0.

- hvClearWeightsFlag - This flag tells CoverBooster to clear dynamically-biased input wieghts. This calls the reset_weight() task in the dynamic_interface implemented in VERA code. This flag is initialized to 0.

- hvStopFlag – This flag causes CoverBooster to call the VCS $stop() routine. This flag is initialized to 1, so that CoverBooster stops after applying the first reset sequence; the Master process then calls $startHV and the actual CTG session can be started.

- hvQuitFlag – This flag causes CoverBooster to exit the main command loop and write out a repeatable test file. VCS then executes the $stop command and waits for further input. The CTG test generation process may not be continued from this point. This flag is initialized to 0.

- hvBiasWeightsFlag – This flag causes CoverBooster to use information in the next replayed trace to modify primary input weighting information. This flag is initialized to 0.

Vera-Accessible PLI Tasks

The CoverBooster program needs access to many of the PLI routines and verilog flags maintained in the verilog testbench top file. The mechanisms VERA uses for this is via verilog "tasks" which are implemented in top and declared external for CoverBooster in an include file. This section describes those VERA-accesible tasks:

- The startHV task calls the $startHV PLI routine and is used by CoverBooster when replaying a previously-generated testcase in stand-

alone mode. More detail on this mode is given in a later section of this document. This task prototype is the following:

```
task startHV;
   input rootInstStr;
   input modeStr;
   input mapStr;
   input inFile;
   input outFile;
   reg [799:0] rootInstStr;
   reg [799:0] modeStr;
   reg [799:0] mapStr;
   reg [799:0] inFile;
   reg [799:0] outFile;
```

- The readFlagsHV task is used to read all control flags in the top module. Task prototype is the following:

```
task readFlagsHV;
   output  resetFlag;
   output  replayTraceFlag;
   output  clearWeightsFlag;
   output  stopFlag;
   output  quitFlag;
   output  biasWeightsFlag;
```

- The following tasks clear the indicated flags and take no arguments.

```
task clearClearWeightsFlagHV;
task clearResetFlagHV;
task clearReplayTraceFlagHV;
task clearStopFlagHV;
task clearQuitFlagHV;
task clearBiasWeightsFlagHV;
```

- The following two tasks make direct calls to the VCS routines $stop and $finish, respectively.

```
task stopVcsHV;
task finishVcsHV;
```

- The getCoverMetricHV task calls the $getCoverMetricHV PLI routine. Task prototype is

```
task getCoverMetricHV;
   input covHandle;
   output  coverValue;
   integer covHandle;
   integer coverValue;
```

- The printCoverInfoHV task calls the $printCoverInfoHV PLI routine. If useRchFlag is non-0, then "COMPACT_REACH" is used, else "COMPACT". A value –1 is used for the input coverage handle. Task prototype is:

```
task printCoverInfoHV;
   input fileNameStr;
   input useRchFlag;
   reg [799:0] fileNameStr;
```

- The readCoverInfoHV task calls the $readCoverInfoHV PLI routine. Task prototype is:

```
task readCoverInfoHV;
   input fileNameStr;
   input useRchFlag;
   reg [799:0] fileNameStr;
```

- The reportDutStateHV task calls the $reportDutStateHV PLI routine. Task prototype is:

```
task reportDutStateHV;
   input fileStr;
   input modeStr;
   reg [799:0] fileStr;
   reg [799:0] modeStr;
```

7.3. Vera Interfacing - CoverBooster

The CoverBooster program is written in VERA and is designed to interface the VCS Slave process with the rest of the CTG system, and to interact with the design-specific VERA code which biases and constrains the random simulation process. This section outlines the design-independent control flow of CoverBooster. User-specified biasing and constraints, compilation methodology, etc are beyond the scope of this document.

Two aspects of CoverBooster are described here: the top-level control flow, and stimulus/sampling top-level clocking strategy.

7.3.1. CoverBooster Clocking Strategy

Clocking strategy in the CoverBooster context deals with when inputs are applied and coverage measured with respect to the SystemClock used with VERA. This is a separate issue from "clocking strategy" for the Design Under Test itself. The DUT clocking strategy might deal with clock dividers, positive/negative registers, edge detectors, etc.

It is expected that part of environmental specification is to synthesize the DUT clock generator in terms of a single SystemClock that CoverBooster knows about.

Given a SystemClock with period T, Figure 4 illustrates when inputs are applied to the DUT and when coverage is sampled.

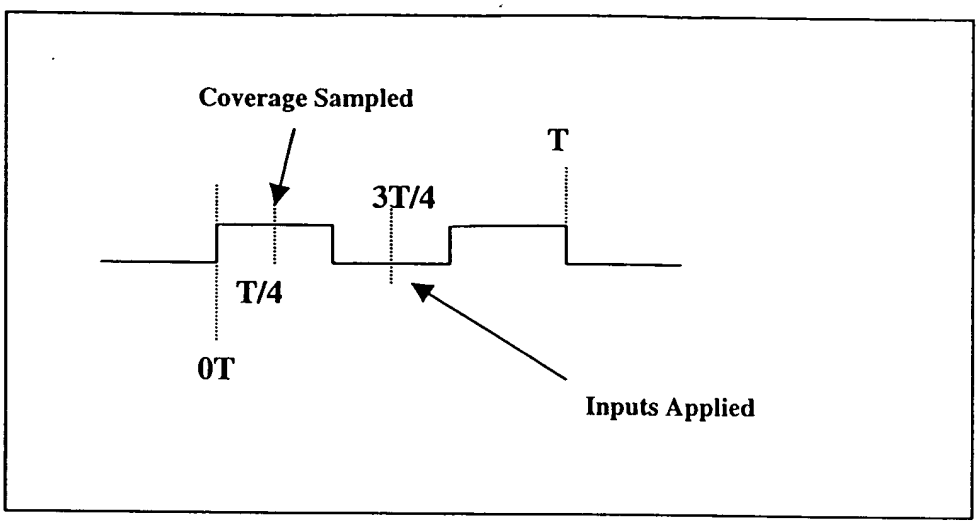


Figure 4. CoverBooster Clocking Strategy

As shown in Figure 4, all inputs are applied ¼ clock period before the positive clock edge. Coverage is measured ¼ cycle after the positive clock edge. Formal search engines are used ¼ cycle before the positive clock edge.

Note that sample points for individual coverage objects may be relative to different clocks in DUT (e.g. see the discussion for the $defineFsmCoverHV() PLI routine). Coverage is measured in CoverBooster to determine when all goals are reached, and to determine if a replayed trace was successful or not.

7.3.2. CoverBooster Control Flow

This section describes key control flow functions in the CoverBooster program.

7.3.2.1. Main CoverBooster Program

Top level control flow is illustrated by the following pseudocode.

```
Program Begin
      initialization and read plus args
      wait for first posedge clock
      wait for next negedge clock
      wait for ¼ more of the clock cycle
      if (testcase replay mode)
            testcaseReplay();
            $finish()
      Endif
      ForEver ()
            ReadControlFlags()
            If (resetFlag)
                  Call interface reset() task
                  ClearResetFlag()
            Elseif (biasWeightsFlag)
                  Remember biasing dynamic request
```

```
            ClearBiasWeightsFlag()
      Elseif (clearWeightsFlag)
            Call resetWeights() in dynamic interface
            ClearClearWeightsFlag()
      Elseif (replayTraceFlag)
            replaySequence()
            Report HIT/MISS to Master
            Setup for dynamic combinational biasing
            ClearReplayTraceFlag()
      Elseif (stopFlag)
            $stop()
            ClearStopFlag()
      Elseif (quitFlag)
            ClearQuitFlag()
            Break
      Else
            If (dynamic biasing)
                  DynamicBiasedDrive()
            Else
                  UserBiasedDrive()
            Endif
            Update coverage metric
            Increment clock counter
            Wait for negedge clock
            Wait for ¼ clock period
      Endif
      If (no goals left)
            Break
      Endif
EndForEver
If (generating test requested)
      Finish writing replay-able test
Endif
$stop()
Program End
```

7.3.2.2. replaySequence VERA Function

This function is called to replay a trace (sequence of inputs to apply over one or more clock cycles). This trace was generated by one of the formal search engines, and should result (usually) in a new goal being hit.

The file which contains the trace information contains all clocking and input assignment information. The replaySequence function simply reads the file and applies logic 1/0 to inputs and waits for clocks as instructed. File format is one token per line, and is the following:

- // - Comment lines begin with two forward slashes
- **!begin!** – This is the first token of the replay trace
- **<portName>** - This token is a primary input port to the Design Under Test. It is followed in the file by
- **<logicValue>** - This token is a '0' or '1' and is the logic value to drive on the port indicated by the previous token.

- **!clock!** – A clock token specifies that a single clock cycle is applied to the design. I.E. one clock period of simulation time advances.

- **!end!** – This token specifies the end of the replay trace.

If requested, dynamic biasing weights are accumulated as the input ports are driven; this information is used later for combinational dynamic biasing.

The following pseudocode illustrates control flow for this function:

```
Function Begin
      OldCover = getCoverageNumber()
      Open trace file
      Text = getNextLineFromFile();
      If (Text != "!begin!")
            Error
      Endif
      ForEver()
            Text = getNextLineFromFile();
            If (Text == "!end!")
                  If (biasingWeights)
                        accumulateWeights();
                  Endif
                  Wait for negedge clock
                  Wait for ¼ clock period
                  Break
            Endif
            If (Text == "!clock!")
                  If (biasingWeights)
                        accumulateWeights();
                  Endif
                  Wait for negedge clock
                  Wait for ¼ clock period
                  Continue
            Endif
            LogicValue = getNextLineFromFile();
            DriveDataOnPort(Text, LogicValue)
      EndForEver

      NewCover = getCoverageNumber()
      If (OldCover > NewCover)
            Report HIT
      Else
            Report MISS
      Endif
Function End
```

7.3.2.3. testcaseReplay VERA Function

This function is called when a complete testcase is to be replayed in stand-alone mode. The test file format is an extension of the trace file format. The test file format, in addition to the tokens described earlier for trace file, is the following:

- **!test_begin!** – This token indicates the begin of a testcase.

- **!run!** – a VCS run should be started. The next line in the file is:

- **<cycleCount>** - specifies the number of clock cycles that should be applied to the design.
- **!reset!** – a reset sequence is to be applied
- **!bias_weights!** – weights should be accumulated when the next trace is replayed.
- **!replay!** – a trace is to be replayed. This token is followed by the token sequence described earlier for trace replay file format.
- **!clear!** – dynamic bias weights should be cleared
- **!test_end!** – This token signifies the end of the testcase

Pseudocode for the testcaseReplay function is the following:

```
Function Begin
      Apply the first reset sequence
      ForEver()
            opCode = getNextLineFromFile();
            If (opCode == "!test_begin!")
                  ReadCoverInfo() // goal definitions
            ElseIf (opCode == "!reset!")
                  Call reset() function
            ElseIf (opCode == "!run!")
                  ClkCnt = getNextLineFromFile();
                  For (i=0;i<clkCnt;i++)
                        If (dynamic comb biasing)
                              Dynamic_biased_drive()
                        Else
                              Random_drive()
                        Endif
                        Wait for negedge clock
                        Wait for ¼ clock period
                  EndFor
            ElseIf (opCode == "!replay!")
                  ReplaySequence()
            ElseIf (opCode == "!clear!")
                  Reset_weights()
            ElseIf (opCode == "!bias_weights!")
                  Next replay biases weights
            ElseIf (opCode == "!test_end!")
                  break
            Else
                  Error
            EndIf
      EndForEver
Function End
```

7.4. Verilog Program Control – Usage Scenarios

There are at least three usage scenarios which the verilog-resident CTG code must support. Verilog "plus args" and PLI routines are used to communicate which mode analysis is intended.

Note: plus args are user-defined verilog invocation switches which are accessible from either PLI routines or from VERA code. Plus args which are defined for use by CTG are the following:

- +ctg_mode=<mode_str> - This plus arg is used to specify which mode VCS is being used for. Currently recognized values are "SLAVE_CREATE", "FREE_REPLAY", and "FREE_CREATE". These are described later in this section.

- +root_dut=<dut_module> - This plus arg specifies which module in verilog is the root for formal search methods.

- +ctg_testfile=<testfile> - In replay mode, this plus arg specifies a testcase file which contains control flow for replay of a previously-created testcase. In test generation mode, this file is written with the generated test.

- +ctg_ingoals=<input_goalfile> - This plus arg specifies an input file which defines coverage goals for the analysis session. This file optionally also contains goals which were covered in previous tests, and are therefore not considered in the current session.

- +ctg_outgoals=<output_goalfile> - This plus arg specifies a file to which reached goal information will be written upon process termination. This file format is suitable for use as <input_goalfile> in subsequent sessions.

The remainder of this section discusses key usage scenarios.

7.4.1. Testcase Creation

This is the classic CTG mode where formal search and explicit state simulation are interleaved to maximize coverage of user-defined goals. Assuming the previously-discussed hv_start_vcs_session TCL procedure is used, invocation of the compiled VCS simulator looks like the following:

```
vcs.exe -s -l vcs.log +ctg_testfile=TestOut \
   +ctg_mode=SLAVE_CREATE
```

This invocation may be slightly different depending on TCL knob settings, and argument used to the hv_start_vcs_session procedure.

The result of this command will be to write a testcase control file "TestOut.ctg", as well as reached goals file "TestOut.vgol" when VCS terminates.

7.4.2. Testcase Replay

Given a testcase created in a previous CTG session, that test may be replayed in stand-alone mode using the following simulator invocation.

```
vcs.exe -s -l vcs.log +ctg_testfile=<testfile> \
   +ctg_mode=FREE_REPLAY +root_dut=<dut_module>
```

Where <testfile> was the testcase filename used in testcase creation. <dut_module> is the verilog root module name for formal search engines during the creation session.

7.4.3. Leveraging User-Written Tests

A common usage for CTG is expected to be that of augmenting hand-written test coverage by use of CTG mechanisms. In this scenario, the hand-written test will cover some goals, and CTG will be used to classify those goals not reached by the hand-written tests. This section describes how this can happen

To collect goals which are reached by the hand-written tests, the following actions are carried out:

1. The user, presumably with some UI help from CTG, defines which goals are of interest; these goals are written to a file in a format readable by the $startHV and $readCoverInfoHV PLI routine.

2. The customer links the CTG PLI library to their VCS simulation.

3. Upon invocation, the user must call the $startHV PLI routine to initiate a CTG collection session. This may either be done using the "–f" verilog switch, or by using VCS interactive mode. The <root_inst> parameter is required to be that which will later be formally searched. <mode> is "FREE_CREATE". <regMapFile> is "". <inGoalFile> is the goalfile created in step 1). <outGoalfile> is specified; <outGoalFile>.ctg and <outGoalFile>.vgol will be written as previously described.

4. If the user has previously reached goals from an earlier sesison, they may be removed from current consideration by calling the $readCoverInfoHV PLI routine.

When this session ends, <outGoalFile>.vgol will contain the goals which were reached in this session, and is suitable as input to a later CTG session.